Common Annotations for the Java™ Platform™

Maintenance Release

Version 1.1

Editor:

Rajiv Mordani

October 2, 2009

Sun Microsystems, Inc.
www.sun.com

August 2009

Submit comments about this document to jsr-250-comments@jcp.org

Specification: JSR-000250 Common Annotations for the Java(tm) Platform ("Specification")

Version: 1.1

Status: Maintenance Release

Release: 24 November 2009

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun'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. Sun 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 Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, 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 Sun'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 Sun 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 Sun 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 Sun 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 Sun 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 Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun'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" or their equivalents in any subsequent naming convention adopted by Sun 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 Sun which corresponds to the Specification and that was available either (i) from Sun'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 Sun if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN 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 SUN 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, IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun 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 Sun 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 Sun 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 th 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.

Rev. April, 2006

Contents

1

Introduction

With the addition of JSR 175 (A Metadata Facility for the Java™ Programming Language) in the Java platform we envision that various technologies will use annotations to enable a declarative style of programming. It would be unfortunate if these technologies each independently defined their own annotations for common concepts. It would be valuable to have consistency within the Java EE and Java SE component technologies, but it will also be valuable to allow consistency between Java EE and Java SE.

It is the intention of this specification to define a small set of common annotations that will be available for use within other specifications. It is hoped that this will help to avoid unnecessary redundancy or duplication between annotations defined in different Java Specification Requests (JSR). This would allow us to have the common annotations all in one place and let the technologies refer to this specification rather than have them specified in multiple specifications. This way all technologies can use the same version of the annotations and there will be consistency in the annotations used across the platforms.

1.1 Goals

Define annotations for use in Java EE 6: This JSR will define annotations for use within component technologies in Java EE 6as well as the platform as a whole.

Define annotations for use in future revisions of Java SE: This JSR will define annotations for use in JSRs targeted for Java SE as well as a future revision of Java SE.

1.2 Non-Goals

Support for Java versions prior to J2SE 5.0

Annotations were introduced in J2SE 5.0. It is not possible to do annotation processing in versions prior to J2SE 5.0. It is not a goal of this specification to define a way of doing annotation processing of any kind for versions prior to J2SE 5.0.

1.3 Compatibility

The annotations defined in this specification may be included individually as needed in products that make use of them. Other Java specifications will require support for subsets of these annotations. Products that support these Java specifications must include the required annotations.

1.4 Conventions

The keywords 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY' AND 'OPTIONAL' in this document are to be interpreted as described in RFC 2119.

Java code is formatted as shown below in figure 1.1:

Figure 1.1 Example Java code

```
package com.wombat.hello;
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

1.5 Expert Group Members

The following expert group members participated in the JSR -

Cedric Beust (individual)

Bill Burke (JBoss)

Wayne Carr (Intel)

Robert Clevenger (Oracle)

Evan Ireland (Sybase)

Woo Jin Kim (Tmax Soft)

Gavin King (JBoss)

Rajiv Mordani (Sun Microsystems, Specification lead)

Ted Neward (individual)

Anurag Parashar (Pramati technologies)

Michael Santos (individual)

Hani Suleiman (Ironflare AB)

Seth White (BEA)

1.6 Acknowledgements

In addition to the expert group listed above the following people Roberto Chinnici, Joe Darcy, Linda DeMichiel, Graham Hamilton, Ron Monzillo, Eduardo Pelegri-Llopart, Scott Seligman and Bill Shannon all of whom work at Sun Microsystems have provided input to this specification.

2

Annotations

This chapter describes the standard annotations, some guidelines for annotation inheritance and the usage of these annotations where possible.

2.1 General Guidelines for Inheritance of Annotations

The interplay of annotations and inheritance in the Java language is potentially a source of complexity for developers. Developers will rely on some implicit assumptions when figuring out how annotations compose with other language features. At the same time, annotation semantics are defined by individual specifications, hence the potential for inconsistencies to arise. For instance, consider the following example:

```
public class Base {
    @TransactionAttribute(REQUIRES_NEW)
    public void foo {....}
}
@Stateless
public class Derived extends Base {
    @TransactionAttribute(NEVER)
    public void foo {....}
}
```

In keeping with the concept of method overriding, most developers will assume that in the `Derived` class, the effective `TransactionAttribute` annotation for method `foo` is `@TransactionAttribute(NEVER)`. On the other hand, it might have been possible for the specification governing the semantics of the `TransactionAttribute` annotations type to require that the effective `TransactionAttribute` to be the most restrictive one in the whole inheritance tree, that is, in the example above `@TransactionAttribute(REQUIRES_NEW)`. A motivation for this semantics might have been that the `foo` method in the `Derived` class may call `super.foo()`, resulting in the execution of some code that needs a transaction to be in place. Such a choice on the part of the specification for `TransactionAttribute` would have contradicted a developer's intuition on how method overriding works.

In order to keep the resulting complexity in control, below are some guidelines recommended for how annotations defined in the different specifications should interact with inheritance:

1. Class-level annotations only affect the class they annotate and their members, that is, its methods and fields. They never affect a member declared by a superclass, even if it is not hidden or overridden by the class in question.

2. In addition to affecting the annotated class, class-level annotations may act as a shorthand for member-level annotations. If a member carries a specific member-level annotation, any annotations of the same type implied by a class-level annotation are ignored. In other words, explicit member-level annotations have priority over member-level annotations implied by a class-level annotation. For example, a `@WebService` annotation on a class implies that all the public methods in the class that it is applied on are annotated with `@WebMethod` if there is no `@WebMethod` annotation on any of the methods. However if there is a `@WebMethod` annotation on any method then the `@WebService` does not imply the presence of `@WebMethod` on the other public methods in the class.

3. The interfaces implemented by a class never contribute annotations to the class itself or any of its members.

4. Members inherited from a superclass and which are not hidden or overridden maintain the annotations they had in the class that declared them, including member-level annotations implied by class-level ones.

5. Member-level annotations on a hidden or overridden member are always ignored.

These set of guidelines guarantees that the effects of an annotation are local to the class on, or inside, which it appears. In order to find the effective annotation for a class member, a developer has to track down its last non-hidden and non-overridden declaration and examine it. If the sought-for annotation is not found there, then (s)he will have to examine the enclosing class declaration. If even this step fails to provide an annotation, no other source file will be consulted.

Below are some examples that explain how the guidelines defined above will be applied to the `TransactionAttribute` annotation.

```
@TransactionAttribute(REQUIRED)
class Base {
    @TransactionAttribute(NEVER)
    public void foo() {...}
    public void bar() {...}
}

@Stateless
class ABean extends Base {
    public void foo() {...}
}

@Stateless
public class BBean extends Base {
    @TransactionAttribute(REQUIRES_NEW)
    public void foo() {...}
}

@Stateless
@TransactionAttribute(REQUIRES_NEW)
public class CBean extends Base {
    public void foo() {...}
    public void bar() {...}
}

@Stateless
@TransactionAttribute(REQUIRES_NEW)
public class DBean extends Base {
    public void bar() {...}
}

@Stateless
@TransactionAttribute(REQUIRES_NEW)
```

```
public class EBean extends Base {
}
```

The table below shows the effective `TransactionAttribute` annotation in each of the cases above by applying the guidelines specified for annotations and inheritance:

TABLE 2-1

Methods in derived classes	Effective TransactionAttribute value
`foo() in ABean`	`REQUIRED (Default TransactionAttribute as defined by the EJB specification).`
`bar() in ABean`	`@Transaction(REQUIRED)`
`foo() in BBean`	`@TransactionAttribute(REQUIRES_NEW)`
`bar() in BBean`	`@TransactionAttribute(REQUIRED)`
`foo() in CBean`	`@TransactionAttribute(REQUIRES_NEW)`
`bar() in CBean`	`@TransactionAttribute(REQUIRES_NEW)`
`foo() in DBean`	`@TransactionAttribute(NEVER) (from the Base class)`
`bar() in DBean`	`@TransactionAttribute(REQUIRES_NEW)`
`foo() in EBean`	`@TransactionAttribute(NEVER) (From Base class)`
`bar() in EBean`	`@TransactionAttribute(REQUIRED)(From Base class)`

For more details about `TransactionAttribute` see the EJB 3 Core Contracts specification.

All annotations defined in this specification follow the guidelines defined above unless explicitly stated otherwise.

Note that even though we use some of the web services annotations in describing the rules above, not all the rules defined in this section are followed by JAX-WS / JSR 181 annotations. Please refer to the JAX-WS / JSR 181 specification for exceptions to these gudelines.

2.2 javax.annotation.Generated

The Generated annotation is used to mark source code that has been generated. It can be specified on a class, methods or fields. It can also be used to differentiate user written code from generated code in a single file. When used, the value element MUST have the name of the code generator. The recommended convention is to use the fully qualified name of the code generator. For example: `com.company.package.classname.` The date element is used to indicate the date the source was generated. The date element MUST follow the ISO 8601 standard. For example the date element would have the following value:

`2001-07-04T12:08:56.235-0700`

which represents 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

The comments element is a place holder for any comments that the code generator may want to include in the generated code.

```
package javax.annotation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target({ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE,
METHOD, PACKAGE, PARAMETER, TYPE})
@Retention(SOURCE)
public @interface Generated {
    String[] value();
    String date() default "";
    String comments() default "";
}
```

TABLE 2-2

Element	Description	Default
`value`	Name of the code generator	
`date`	Date source was generated. MUST follow ISO 8601 standard	“”
`comments`	placeholder for comments that the generator may want to include in the generated code	“”

The following example shows the usage of the annotation defined above:

```
@Generated(“com.sun.xml.rpc.AProcessor”)
public interface StockQuoteService extends java.rmi.Remote {
    this.context = context;
}
```

2.3 javax.annotation.Resource

The Resource annotation is used to declare a reference to a resource. It can be specified on a class, methods or on fields. When the annotation is applied on a field or method, the container will inject an instance of the requested resource into the application when the application is initialized. If the annotation is applied to a class, the annotation declares a resource that the application will look up at runtime. Even though this annotation is not marked `Inherited`, if used all superclasses MUST be examined to discover all uses of this annotation. All such annotation instances specify resources that are needed by the application. Note that this annotation may appear on private fields and methods of the superclasses. Injection of the declared resources needs to happen in these cases as well, even if a method with such an annotation is overridden by a subclass.

The name element is the JNDI name of the resource. When the Resource annotation is applied on a field, the default value of the name element is the field name qualified by the class name. When applied on a method, the default is the JavaBeans property name corresponding to the method qualified by the class name. When applied on a class, there is no default and the name MUST be specified.

The type element defines the Java type of the resource. When the Resource annotation is applied on a field, the default value of the type element is the type of the field. When applied on a method, the default is the type of the JavaBeans property. When applied on a class, there is no default and the type MUST be specified. When used, the type MUST be assignment compatible.

The authenticationType element is used to indicate the authentication type to use for the resource. It can take one of two values defined as an Enum: CONTAINER or APPLICATION. This element may be specified for resources representing a connection factory of any supported type and MUST NOT be specified for resources of other types.

The shareable element is used to indicate whether a resource can be shared between this component and other components. This element may be specified for resources representing a connection factory of any supported type or ORB object instances and MUST NOT be specified for resources of other types.

The mappedName element is a product specific name that this resource should be mapped to. The name of this resource, as defined by the name element or defaulted, is a name that is local to the application using the resource. Many application servers provide a way to map these local names to names of resources known to the application server. The mapped name could be of any form. Application servers are not required to support any particular form or type of mapped name, nor the ability to use mapped names. The mapped name is product-dependent and often installation-dependent. No use of mapped name is portable.

The description element is the description of the resource. The description is expected to be in the default language of the system on which the application is deployed. The description can be presented to help in choosing the correct resource.

The lookup element specifies the JNDI name of a resource that the resource being defined will be bound to. The type of the referenced resource must be compatible with that of the resource being defined.

```
package javax.annotation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Resource {
```

```
    public enum AuthenticationType {
        CONTAINER,
        APPLICATION
    }
    String name() default "";
    Class type() default Object.class;
    AuthenticationType authenticationType() default
                          AuthenticationType.CONTAINER;
    boolean shareable() default true;
    String mappedName() default "";
    String description() default "";
    String lookup() deafult "";
}
```

TABLE 2-3

Element	Description	Default
`name`	The JNDI name of the resource	""
`type`	The Java type of the resource	`Object.class`
`authenticationType`	The authentication type to use for the resource	`CONTAINER`
`shareable`	Indicates whether the resource can be shared.	`true`
`mappedName`	A product specific name that the resource should map to.	""
`description`	Description of the resource.	""
`lookup`	the JNDI name of a resource that the resource being defined will be bound to	""

Field based injection:

To access a resource a developer declares a field and annotates it as being a resource reference. If the name and type elements are missing from the annotation it will be inferred by looking at the field declaration itself. It is an error if the type specified by the `@Resource` annotation and the type of the field are incompatible.

For example:

```
@Resource
private DataSource myDB;
```

In the example above the effective name is `com.example.class/myDB` and the effective type is `javax.sql.DataSource.class`.

```
@Resource(name="customerDB")
private DataSource myDB;
```

In the example above the name is `customerDB` and the effective type is `javax.sql.DataSource.class`.

Setter based injection:

To access a resource a developer declares a setter method and annotates it as being a resource reference. The name and type of resource may be inferred by inspecting the method declaration if necessary. The name of the resource, if not declared, is the name of the JavaBeans property as determined from the name of the setter method in question. The setter method MUST follow the standard JavaBeans convention - name starts with a "set", `void` return type, and only one parameter. Additionally, the type of the parameter MUST be compatible with the type specified as a property of the Resource, if present.

For example:

```
@Resource
private void setMyDB(DataSource ds) {
    myDB = ds;
}
private DataSource myDB;
```

In the example above the effective name is `com.example.class/myDB` and the type is `javax.sql.DataSource.class`.

```
@Resource(name="customerDB")
private void setMyDB(DataSource ds) {
    myDB = ds;
}
private DataSource myDB;
```

In the example above the name is `customerDB` and the type is `javax.sql.DataSource.class`.

The table below shows the mapping from Java type to the equivalent resource type in Java EE 5 deployment descriptors:

TABLE 2-4

Java Type	Equivalent Resource type
java.lang.String	env-entry
java.lang.Character	env-entry
java.lang.Integer	env-entry
java.lang.Boolean	env-entry
java.lang.Double	env-entry
java.lang.Byte	env-entry
java.lang.Short	env-entry
java.lang.Long	env-entry
java.lang.Float	env-entry
javax.xml.rpc.Service	service-ref
javax.xml.ws.Service	service-ref
javax.jws.WebService	service-ref
javax.sql.DataSource	resource-ref
javax.jms.ConnectionFactory	resource-ref
javax.jms.QueueConnectionFactory	resource-ref
javax.jms.TopicConnectionFactory	resource-ref
javax.mail.Session	resource-ref
java.net.URL	resource-ref
javax.resource.cci.ConnectionFactory	resource-ref
org.omg.CORBA_2_3.ORB	resource-ref
any other connection factory defined by a resource adapter	resource-ref
javax.jms.Queue	message-destination-ref
javax.jms.Topic	message-destination-ref

Java Type	Equivalent Resource type
javax.resource.cci.InteractionSpec	resource-env-ref
javax.transaction.UserTransaction	resource-env-ref
Everything else	resource-env-ref

2.4 javax.annotation.Resources

The `Resource` annotation is used to declare a reference to a resource. Since repeated annotations are not allowed, the `Resources` annotation acts as a container for multiple resource declarations.

```
package javax.annotation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target({TYPE})
@Retention(RUNTIME)
public @interface Resources {
    Resource[] value;
}
```

TABLE 2-5

Element	Description	Default
`value`	Container for defining multiple resources.	

The following example shows the usage of the annotation defined above:

```
@Resources ({
    @Resource(name="myDB" type=javax.sql.DataSource),
    @Resource(name="myMQ" type=javax.jms.ConnectionFactory)
})
public class CalculatorBean {
```

```
    //...
}
```

2.5 javax.annotation.PostConstruct

The `PostConstruct` annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization. This method MUST be invoked before the class is put into service. This annotation MUST be supported on all classes that support dependency injection. The method annotated with `PostConstruct` MUST be invoked even if the class does not request any resources to be injected. Only one method can be annotated with this annotation. The method on which the `PostConstruct` annotation is applied MUST fulfill all of the following criteria:

- The method MUST NOT have any parameters except in the case of EJB interceptors in which case it takes an `InvocationContext` object as defined by the EJB specification.

- The return type of the method MUST be `void`.

- The method MUST NOT throw a checked exception.

- The method on which `PostConstruct` is applied MAY be `public`, `protected`, package private or `private`.

- The method MUST NOT be static except for the application client.

- The method MAY be final or non-final, except in the case of EJBs where it MUST be non-final.

- If the method throws an unchecked exception the class MUST NOT be put into service. In the case of EJBs the method annotated with `PostConstruct` can handle exceptions and cleanup before the bean instance is discarded.

```
package javax.annotation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target(METHOD)
@Retention(RUNTIME)
public @interface PostConstruct {
}
```

The following example shows the usage of the annotation defined above:

```
@Resource
private void setMyDB(DataSource ds) {
    myDB = ds;
}

@PostConstruct
private void initialize() {
    //Initialize the connection object from the DataSource
    connection = myDB.getConnection();
}

private DataSource myDB;
private Connection connection;
```

2.6 javax.annotation.PreDestroy

The `PreDestroy` annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container. The method annotated with `PreDestroy` is typically used to release resources that it has been holding. This annotation MUST be supported by all container managed objects that support `PostConstruct` except the application client container in Java EE 5. The method on which `the PreDestroy` annotation is applied MUST fulfill all of the following criteria:

- The method MUST NOT have any parameters except in the case of EJB interceptors in which case it takes an `InvocationContext` object as defined by the EJB specification.

- The return type of the method MUST be `void`.

- The method MUST NOT throw a checked exception.

- The method on which `PreDestroy` is applied MAY be `public`, `protected`, package private or `private`.

- The method MUST NOT be static.

- The method MAY be final or non-final, except in the case of EJBs where it MUST be non-final.
- If the method throws an unchecked exception it is ignored except in the case of EJBs where the method annotated with `PreDestroy` can handle exceptions.

```
package javax.annotation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target(METHOD)
@Retention(RUNTIME)
public @interface PreDestroy {
}
```

The following example shows the usage of the annotation defined above:

```
@Resource
private void setMyDB(DataSource ds) {
    myDB = ds;
}

@PostConstruct
private void initialize() {
    //Initialize the connection object from the DataSource
    connection = myDB.getConnection();
}
@PreDestroy
private void cleanup() {
    //Close the connection to the DataSource.
    connection.close();
}

private DataSource myDB;
private Connection connection;
```

2.7 javax.annotation.security.RunAs

The `RunAs` annotation defines the role of the application during execution in a Java EE container. It can be specified on a class. This allows developers to execute an application under a particular role. The role MUST map to the user / group information in the container's security realm. The `value` element in the annotation is the name of a security role.

```
package javax.annotation.security;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target(TYPE)
@Retention(RUNTIME)
public @interface RunAs {
    String value();
}
```

TABLE 2-6

Element	Description	Default
`value`	Security role of the application during execution in a Java EE container	

The following example shows the usage of the annotation defined above:

```
@RunAs("Admin")
public class Calculator {
    //....
}
```

2.8 javax.annotation.security.RolesAllowed

The `RolesAllowed` annotation specifies the security roles permitted to access method(s) in an application. The `value` element of the `RolesAllowed` annotation is a list of security role names.

The `RolesAllowed` annotation can be specified on a class or on method(s). Specifying it at a class level means that it applies to all the methods in the class. Specifying it on a method means that it is applicable to that method only. If applied at both the class and method level, the method value overrides the class value.

```
package javax.annotation.security;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target({TYPE,METHOD})
@Retention(RUNTIME)
public @interface RolesAllowed {
    String[] value();
}
```

TABLE 2-7

Element	Description	Default
`value`	List of roles permitted to access methods in the application	

The following example shows the usage of the annotation defined above:

```
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    //..
}
```

2.9 javax.annotation.security.PermitAll

The `PermitAll` annotation specifies that all security roles are allowed to invoke the specified method(s), that is, that the specified method(s) are "unchecked". It can be specified on a class or on methods. Specifying it on the class means that it applies to all methods of the class. If specified at the method level, it only affects that method.

```
package javax.annotation.security;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target({TYPE,METHOD})
@Retention(RUNTIME)
public @interface PermitAll {
}
```

The following example shows the usage of the annotation defined above:

```
import javax.annotation.security.*;
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
    @PermitAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

2.10 javax.annotation.security.DenyAll

This annotation specifies that no security roles are allowed to invoke the specified method(s), that is, that the method(s) are to be excluded from execution in the Java EE container.

```
package javax.annotation.security;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface DenyAll {
}
```

The following example shows the usage of the annotation defined above:

```
import javax.annotation.security.*;
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed(“Administrator”)
    public void setNewRate(int rate) {
        //...
    }
    @DenyAll
    public long convertCurrency(long amount) {
        //...
    }
}
```

2.11 PermitAll, DenyAll and RolesAllowed interactions

The `PermitAll`, `DenyAll` and `RolesAllowed` annotations all define what security roles are allowed to access the methods on which they are applied. This section describes how these annotations interact and which usages of these annotations are valid.

If the `PermitAll`, `DenyAll` and `RolesAllowed` annotations are applied on methods of a class, then the method level annotations take precedence (at the corresponding methods) over any class level annotations of type `PermitAll`, `DenyAll` and `RolesAllowed`.

2.12 javax.annotation.security.DeclareRoles

This annotation is used to specify the security roles by the application. It can be specified on a class. It typically would be used to define roles that could be tested (i.e., by calling `isUserInRole`) from within the methods of the annotated class. It could also be used to declare roles that are not implicitly declared as the result of their use in a `RolesAllowed` annotation on the class or a method of the class.

```
package javax.annotation.security;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target(TYPE)
@Retention(RUNTIME)
public @interface DeclareRoles{
    String[] value();
}
```

TABLE 2-8

Element	Description	Default
`value`	List of security roles specified by the application	

The following example shows the usage of the annotation defined above:

```
@DeclareRoles("BusinessAdmin")
public class Calculator {
    public void convertCurrency() {
        if(x.isUserInRole("BusinessAdmin")) {
          //....
        }
    }
    //...
}
```

2.13 javax.annotation.sql.DataSourceDefinition

This annotation is used to define a container `DataSource` and be registered with JNDI. The `DataSource` may be configured by setting the annotation elements for commonly used `DataSource` properties. Additional standard and vendor-specific properties may be specified using the properties element. The data source will be registered under the name specified in the `name` element. It may be defined to be in any valid Java EE namespace, and will determine the accessibility of the data source from other components. A JDBC driver implementation class of the appropriate type, either `DataSource`, `ConnectionPoolDataSource`, or `XADataSource`, must be indicated by the `className` element. The driver class is not required to be available at deployment but must be available at runtime prior to any attempt to access the `DataSource`.

The `url` property should not be specified in conjunction with other standard properties for defining the connectivity to the database. If the `url` property is specified along with other standard `DataSource` properties such as `serverName` and `portNumber`, the more specific properties will take precedence and `url` will be ignored.

Vendors are not required to support `properties` that do not normally apply to a specific data source type. For example, specifying the `transactional` property to be `true` but supplying a value for `className` that implements a data source class other than `XADataSource` may not be supported.

Vendor-specific properties may be combined with or used to override standard data source properties defined using this annotation.

`DataSource` properties that are specified and are not supported in a given configuration or cannot be mapped to a vendor specific configuration property may be ignored.

Although the annotation allows you to specify a password, it is recommended not to embed passwords in production code. The password element in the annotation is provided as a convenience for ease of development.

```
package javax.annotation.sql;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DataSourceDefinition {
    String name();
    String className();
    String description() default "";
    String url() default "";
    String user() default "";
    String password() default "";
    String databaseName() default "";
    int portNumber() default -1;
    String serverName() default "localhost";
    int isolationLevel() default -1;
    boolean transactional() default true;
    int initialPoolSize() default -1;
    int maxPoolSize() default -1;
    int minPoolSize() default -1;
    int maxIdleTime() default -1;
    int maxStatements() default -1;
    String[] properties() default {};
    int loginTimeout() default 0;
}
```

TABLE 2-9

Element	Description	Default
`name`	JNDI name by which the data source will be registered	
`className`	DataSource implementation class name	
`description`	Description of the data source	""
`url`	A JDBC url. If the url property is specified along with other standard DataSource properties such as serverName and portNumber, the more specific properties will take precedence and the url will be ignored	""
`user`	User name for connection authentications	""
`password`	password for connection authentications	""
`databaseName`	Name of a database on a server	""
`portNumber`	Port number where a server is listening for requests	""
`serverName`	Database server name	"localhost"
`isolationLevel`	Isolation level for connections.	-1 (vendor specific)
`transactional`	Indicates whether a connection is transactional or not	true
`initialPoolSize`	Number of connections that should be created when a connection pool is initialized	-1 (vendor specific)
`maxPoolSize`	Maximum number of connections that should be concurrently allocated for a connection pool	-1 (vendor specific)
`minPoolSize`	Minimum number of connections that should be allocated for a connection pool	-1 (vendor specific)
`maxIdleTime`	The number of seconds that a physical connection should remain unused in the pool before the connection is closed for a connection pool	-1 (vendor specific)

Element	Description	Default
`maxStatements`	The total number of statements that a connection pool should keep open. A value of 0 indicates that the caching of statements is disabled for a connection pool	-1 (vendor specific)
`properties`	Used to specify vendor specific properties and less commonly used DataSource properties	{}
`loginTimeout`	The maximum time in seconds that this data source will wait while attempting to connect to a database. A value of 0 specifies that the timeout is the default system timeout if there is one, otherwise it specifies that there is no timeout	0

Examples:

```
@DataSourceDefinition(name="java:global/MyApp/MyDataSource",
    className="com.foobar.MyDataSource",
    portNumber=6689,
    serverName="myserver.com",
    user="lance",
    password="secret")
```

Using a URL:

```
@DataSourceDefinition(name="java:global/MyApp/MyDataSource",
    className="org.apache.derby.jdbc.ClientDataSource",
    url="jdbc:derby://localhost:1527/myDB",
    user="lance",
    password="secret")
```

2.14 javax.annotation.sql.DataSourceDefinitions

The `DataSource` annotation is used to declare a container `DataSource`. Since repeated annotations are not allowed, the `DataSourceDefinitions` annotation acts as a container for multiple data source declaration.

```
package javax.annotation.sql;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DataSourceDefinitions {
    DataSourceDefinition[] value ();
}
```

TABLE 2-10

Element	Description	Default
`value`	Container for defining multiple data sources.	

The following example shows the usage of the annotation defined above:

```
@DataSourceDefinitions ({
@DataSourceDefinition(name="java:global/MyApp/MyDataSource",
    className="com.foobar.MyDataSource",
    portNumber=6689,
    serverName="myserver.com",
    user="lance",
    password="secret")
@DataSourceDefinition(name="java:global/MyApp/MyDataSource",
    className="org.apache.derby.jdbc.ClientDataSource",
```

```
    url="jdbc:derby://localhost:1527/myDB",
    user="lance",
    password="secret")
})
public class CalculatorBean {
    //...
}
```

2.15 javax.annotation.ManagedBean

This annotation is used to declare a Managed Bean as specified in the Managed Beans specification. Managed Beans are container managed objects that support a small set of basic services such as resource injection, lifecycle callbacks and interceptors. A Managed Bean may optionally have a `name`, a `String` specified via the `value` element.

```
package javax.annotation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
@Target(TYPE)
@Retention(RUNTIME)
public @interface ManagedBean {
     boolean value() default "";
}
}
```

TABLE 2-11

Element	Description	Default
`value`	Name of the Managed Bean	""

Examples:

```
@ManagedBean("cart")
```

```
public class ShoppingCart {
...

}
```

CHAPTER 3

References

1. JSR 175: A Metadata Facility for the Java Programming Language. *http://jcp.org/en/jsr/detail?id=175*
2. Java Platform, Enterprise Edition, v6 (Java EE). *http://jcp.org/en/jsr/detail?id=316*
3. Java 2 Platform, Standard Edition, v5.0 (J2SE). *http://java.sun.com/j2se*
4. Enterprise JavaBeans, v3.1 (EJB). *http://jcp.org/en/jsr/detail?id=318*
5. RFC 2119. *http://www.faqs.org/rfcs/rfc2119.html*