Sun Microsystems

JSR 220: Enterprise JavaBeans™,Version 3.0

EJB 3.0 Simplified API

EJB 3.0 Expert Group

Specification Lead:
Linda DeMichiel, Sun Microsystems
Michael Keith, Oracle Corporation

Please send comments to: ejb3-pfd-feedback@sun.com

Version 3.0, Proposed Final Draft

December 18, 2005

Specification: JSR-000220 Enterprise JavaBeans(tm) v.3.0 ("Specification")

Status: Pre-FCS, Proposed Final Draft
Release: December 21, 2005

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

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 TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from 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: If this Software 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 Specification 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: You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any 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 related to the Specification and future versions, implementations, and test suites thereof.

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

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

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

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.

(Sun.pre-FCS.Spec.license.11.14.2003)

Table of Contents

Chapter 1 Introduction

1.1 Overview

The EJB 3.0 release of the Enterprise JavaBeans architecture provides a new, simplified API for the enterprise application developer. This API is targeted at ease of development and is a simplification of the APIs defined by earlier versions of the EJB specification. The existing EJB 2.1 APIs remain available for use in applications that require them and components written to those APIs may be used in conjunction with components written to the new EJB 3.0 APIs.

This document provides an overview of the EJB 3.0 simplified API.

1.2 Goals of this Release

The purpose of the EJB 3.0 release is to improve the EJB architecture by reducing its complexity from the enterprise application developer's point of view.

EJB 3.0 is focused on the following goals:

- *Definition of the Java language metadata annotations that can be used to annotate EJB applications. These metadata annotations are targeted at simplifying the developer's task, at reducing the number of program classes and interfaces the developer is required to implement, and at eliminating the need for the developer to provide an EJB deployment descriptor.*
- *Specification of programmatic defaults, including for metadata, to reduce the need for the developer to specify common, expected behaviors and requirements on the EJB container. A "configuration by exception" approach is taken whenever possible.*
- *Encapsulation of environmental dependencies and JNDI access through the use of annotations, dependency injection mechanisms, and simple lookup mechanisms.*
- *Simplification of the enterprise bean types.*
- *Elimination of the requirement for EJB component interfaces for session beans. The required business interface for a session bean can be a plain Java interface rather than an EJBObject, EJBLocalObject, or java.rmi.Remote interface.*
- *Elimination of the requirement for home interfaces for session beans.*
- *Simplification of entity bean persistence. Support for light-weight domain modeling, including inheritance and polymorphism.*

- *Elimination of all required interfaces for persistent entities [2].*
- *Specification of Java language metadata annotations and XML deployment descriptor elements for the object/relational mapping of persistent entities [2].*
- *Enhancements to EJB QL to provide additional functionality. Addition of projection, explicit inner and outer join operations, bulk update and delete, subqueries, and group-by. Addition of a dynamic query capability and support for native SQL queries.*
- *An interceptor facility for session beans and message-driven beans.*
- *Reduction of the requirements for usage of checked exceptions.*
- *Elimination of the requirement for the implementation of callback interfaces.*

1.3 EJB 3.0 Expert Group

The EJB 3.0 specification work is being conducted as part of JSR-220 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the EJB 3.0 Expert Group. These include the following present and former expert group members: Apache Software Foundation: Jeremy Boynes; BEA: Seth White; Borland: Jishnu Mitra; E.piphany: Karthik Kothandaraman; Fujitsu-Siemens: Anton Vorsamer; Google: Cedric Beust; IBM: Jim Knutson, Randy Schnier; IONA: Conrad O'Dea; Ironflare: Hani Suleiman; JBoss: Gavin King, Bill Burke, Marc Fleury; Macromedia: Hemant Khandelwal; Nokia: Vic Zaroukian; Novell: YongMin Chen; Oracle: Michael Keith, Debu Panda, Olivier Caudron; Pramati: Deepak Anupalli; SAP: Steve Winkler, Umit Yalcinalp; SAS Institute: Rob Saccoccio; SeeBeyond: Ugo Corda; SolarMetric: Patrick Linskey; Sun Microsystems: Linda DeMichiel, Mark Reinhold; Sybase: Evan Ireland; Tibco: Shivajee Samdarshi; Tmax Soft: Woo Jin Kim; Versant: David Tinker; Xcalia: Eric Samson; Reza Behforooz; Emmanuel Bernard; Wes Biggs; David Blevins; Scott Crawford; Geoff Hendrey; Oliver Ihns; Oliver Kamps; Richard Monson-Haefel; Dirk Reinshagen; Carl Rosenberger; Suneet Shah.

1.4 Organization of the Specification Documents

This specification is organized into the following three documents:

- EJB 3.0 Simplified API
- EJB Core Contracts and Requirements
- Java Persistence

The current document provides an overview of the simplified API that is introduced by the Enterprise JavaBeans 3.0 release.

The document "*Java Persistence*" is the specification of the new API for the management of persistence together with the full specification of EJB QL. It provides the definition of the persistence API that is required to be supported under the Enterprise JavaBeans 3.0 release as well as the definition of how Java Persistence is supported for use in Java SE environments.

The document "*EJB Core Contracts and Requirements*" defines the contracts and requirements for the use and implementation of Enterprise JavaBeans. These contracts include, by reference, those defined in the "*Java Persistence*" document.

1.5 Document Conventions

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

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

```
The Courier font is used for code examples.
```

Chapter 2 Overview of the EJB 3.0 Simplified API

The EJB 3.0 release is focused on a simplification of the Enterprise JavaBeans architecture from the developer's point of view.

This simplification has several main aspects:

- Simplification of the interface definition requirements for enterprise beans: elimination of requirements for the specification of home and component interfaces in the EJB 3.0 programming model.
- Simplification of the contractual requirements between the bean provider and the container: elimination of the requirements for enterprise beans to implement the `javax.ejb.EnterpriseBean` interfaces.
- Simplification of APIs for access to a bean's environment: definition of a dependency injection facility and simpler look-up APIs.
- Introduction of Java metadata annotations to be used as an alternative to deployment descriptors.
- Simplification of object persistence by the definition of a light-weight object/relational mapping facility based on the direct use of Java classes rather than persistent components.

2.1 Metadata Annotations and Deployment Descriptors

One of the key enabling technologies introduced by J2SE 5.0 is the program annotation facility defined by JSR-175 [10]. This facility allows developers to annotate program elements in Java programming language source files to control the behavior and deployment of an application.

Metadata annotations are a key element in the simplification of the development of EJB 3.0 applications. Metadata annotations are used by the developer to specify expected requirements on container behavior, to request the injection of services and resources, and to specify object/relational mappings.

Metadata annotations may be used as an alternative to the deployment descriptors that were required by earlier versions of the Enterprise JavaBeans specification.

While this document is written in terms of the usage of metadata annotations, it is not required that metadata annotations be used in an EJB 3.0 application. Developers who prefer the use of a deployment descriptor as an alternative to metadata annotations may define one for this purpose. The EJB 3.0 deployment descriptor is defined in the document "EJB Core Contracts and Requirements" [1] of this specification.

2.1.1 Deployment Descriptors

Deployment descriptors are defined by this specification as an alternative to metadata annotations or as a mechanism for the overriding of metadata annotations—for example to permit the further customization of an application for a particular development environment at a later stage of the development or assembly work flow. Deployment descriptors may be "sparse", unlike the full deployment descriptors required by the EJB 2.1 specification. See "*EJB Core Contracts and Requirements*" [1].

Although it is not anticipated as a typical use case, it is possible for the application developer to combine the use of metadata annotations and deployment descriptors in the design of an application. When such a combination is used, the rules for the use of deployment descriptors as an overriding mechanism apply.

2.2 Interoperability and Migration Between EJB 3.0 and EJB 2.1 and Earlier Clients and Beans

A bean written to the EJB 3.0 APIs may be a client of components written to the EJB 2.1 and earlier APIs, and vice versa. Chapter 9 "Compatibility and Migration" describes the mechanisms and APIs that enable this.

Such combinations of clients and components written to different versions of the Enterprise JavaBeans specification programming models may be useful in facilitating the migration of existing applications incrementally to EJB 3.0, for adding new functionality to applications written to earlier versions of the Enterprise JavaBeans specification, and for reuse of components and applications written to the earlier EJB APIs.

Chapter 3

Enterprise Bean Class and Business Interface

This chapter describes aspects of the EJB 3.0 programming model that are common across session bean and message-driven bean component types.[1]

3.1 Enterprise Bean Class

In programming with the EJB 3.0 API, the developer typically uses the enterprise bean class as the primary programming artifact.

The bean developer defines the enterprise bean class and annotates it using the Java metadata annotations defined by this and related specifications [7], [8], [9], [11]. Metadata annotations may be applied to the bean class to specify semantics and requirements to the EJB container, to request container services, and/or to provide structural and configuration information to the application deployer or the container runtime. (See Chapter 10 “Metadata Annotations”).

[1] The persistent entities defined in the document “*Java Persistence*” [2] of this specification—unlike EJB 2.1 entity beans—are not enterprise bean components. The contracts described in this specification document therefore do not apply to them.

3.1.1 Requirements for the Enterprise Bean Class

The bean type of the enterprise bean class must be specified. The bean type is typically specified by means of metadata annotations. Deployment descriptor elements may be used as an alternative.

Example

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

3.2 Business Interfaces

Under the EJB 3.0 API, the business interface of an enterprise bean is a plain Java interface, not an EJBObject or EJBLocalObject interface. [2]

Session beans and message-driven beans require a business interface. The business interface of a message-driven bean is typically defined by the messaging type used (e.g., `javax.jms.MessageListener` in the case of JMS). Business interfaces in the sense of this chapter are not defined for entity beans.

The bean class may implement its business interface(s).[3] A bean class may have more than one business interface. The following rules apply:

- If bean class implements a single interface, that interface is assumed to be the business interface of the bean. This business interface will be a local interface unless the interface is designated as a remote business interface by use of the `Remote` annotation on the bean class or interface or by means of the deployment descriptor.

- A bean class is permitted to have more than one interface. If a bean class has more than one interface—excluding the interfaces listed below—any business interface of the bean class must be explicitly designated as a business interface of the bean by means of the `Local` or `Remote` annotation on the bean class or interface or by means of the deployment descriptor.

  The following interfaces are excluded when determining whether the bean class has more than one interface: `java.io.Serializable`; `java.io.Externalizable`; any of the interfaces defined by the `javax.ejb` package.

- A business interface must not extend `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject`.

[2] Usage of the earlier EJBObject and EJBLocalObject interface types continues to be supported under EJB 3.0. See Chapter 9 “Compatibility and Migration” .

[3] While it is expected that the bean class will typically implement its business interface(s), if the bean class uses annotations on the bean class or the deployment descriptor to designate its business interface(s), it is not required that the bean class also be specified as implementing the interface(s). See the document “*EJB Core Contracts and Requirements*” [1].

The metadata annotations to specify that a bean implements a web service and how the web service is exposed to clients are defined by JSR-181, "Web Services Metadata for the Java Platform."[9]

Example

```
@Stateless @Remote
public class CalculatorBean implements Calculator {

  public float add (int a, int b) {
    return a + b;
  }

  public float subtract (int a, int b) {
    return a - b;
  }
}

public interface Calculator {
   public float add (int a, int b);
   public float subtract (int a, int b);
  }
```

Example

```
// Shopping Cart is the local business interface of the bean

@Stateful public class ShoppingCartBean implements ShoppingCart {
     ...

}
```

3.3 Exceptions

The methods of the business interface may declare arbitrary application exceptions. However, the methods of the business interface should not throw the `java.rmi.RemoteException`, even if the interface is a remote business interface, the bean class is annotated `WebService`, or the method is annotated as `WebMethod` (see [9]). If problems are encountered at the protocol level, an EJBException which wraps the underlying RemoteException is thrown by the container. See the specification document "*EJB Core Contracts and Requirements*" [1], Chapter 13 "Exception Handling".

3.4 Interceptors

An interceptor is a method that intercepts a business method invocation or a lifecycle callback event. An interceptor method may be defined on the bean class or on an interceptor class associated with the bean. An interceptor class is a class (distinct from the bean class itself) whose methods are invoked in response to business method invocations and/or lifecycle events on the bean class. Interceptors may be

defined for session beans and message-driven beans.

Interceptor classes are denoted using the `Interceptors` annotation on the bean class with which they are associated. Default interceptors—interceptors that apply to all session beans and message driven beans in the ejb-jar file—may be defined by means of the deployment descriptor.

Any number of interceptor classes may be defined for a bean class. If multiple interceptors are defined, the order in which they are invoked is determined by the order in which they are specified. (See "*EJB Core Contracts and Requirements*".)

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

Interceptors are stateless. The lifecycle of an interceptor is unspecified, as is the sharing of interceptors across multiple threads. Dependency injection is performed when the interceptor instance is created, using the naming context of the associated enterprise bean.

It is possible to carry state across multiple interceptor method invocations for a single interceptor method invocation on a bean in the context data of the `InvocationContext` object.

Interceptors are statically configured by annotations or in the deployment descriptor.

The following rules apply to interceptors. The full set of requirements for interceptors is defined in the document "*EJB Core Contracts and Requirements*" of this specification.

- Business method interceptor method invocations occur within the same transaction and security context as the business method for which they are invoked.
- Business method interceptor methods may throw runtime exceptions or application exceptions that are allowed in the throws clause of the business method. Lifecycle callback interceptor methods may throw runtime exceptions.
- Interceptors can invoke JNDI, JDBC, JMS, other enterprise beans, and the EntityManager. See "*EJB Core Contracts and Requirements*" [1], Tables 1, 2, 3. Interceptor methods share the JNDI name space of the bean for which they are invoked.
- Dependency injection is supported for interceptor classes.
- Programming restrictions that apply to enterprise bean components to apply to interceptors as well. See "*EJB Core Contracts and Requirements*", Section 20.1.2 [1].]

3.4.1 Lifecycle Callback Interceptor Methods

A method may be designated as a lifecycle callback interceptor method to receive notification of life cycle events for a session bean or message-driven bean. Lifecycle callback interceptor methods are annotated with the `PostConstruct`, `PreDestroy`, `PostActivate`, or `PrePassivate` annotations.

Example

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

An interceptor class may be used instead of callback methods defined directly on the bean class.

Lifecycle callback methods on the bean class or on the interceptor class are statically configured for a bean class by use of metadata annotations or the deployment descriptor.

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

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

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

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

The annotations used for lifecycle callback interceptor methods on the bean class and on the interceptor class are the same. The same method may be annotated with more than one callback annotation, thus serving for more than one callback.

Any subset or combination of lifecycle callback annotations may be specified on the bean class or on an associated interceptor class. The same callback may not be specified more than once on a given class.

The requirements for lifecycle callback methods and interceptors are described further in the document "*EJB Core Contracts and Requirements*" of this specification [1].

3.4.2 Business Method Interceptor Methods

Business method interceptor methods may be defined for session bean business methods and the message listener methods of message-driven beans. Business method interceptor methods are denoted by the `AroundInvoke` annotation. Only one `AroundInvoke` method may be present on the bean class or on any given interceptor class. An `AroundInvoke` method must not be a business method.

The business method invocation is intercepted by the `AroundInvoke` methods of the bean class and interceptor classes. `AroundInvoke` methods must always call `InvocationContext.proceed()` or neither the business method will be invoked nor any subsequent interceptor `AroundInvoke` methods.

`AroundInvoke` methods have the following signature:

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

3.4.3 InvocationContext

The `InvocationContext` object provides the metadata that is required for interceptor methods:

```
public interface InvocationContext {
     public Object getBean();
     public Method getMethod();
     public Object[] getParameters();
     public void setParameters(Object[]);
     public java.util.Map getContextData();
     public Object proceed() throws Exception;
}
```

The same `InvocationContext` instance is passed to each interceptor method for a given business method interception or lifecycle event. This allows an interceptor to save information in the context data property of the `InvocationContext` that can be subsequently retrieved in other interceptors as a means to pass contextual data between interceptors. The contextual data is not shareable across business method invocations or lifecycle event callbacks. The lifecycle of the `InvocationContext` instance is otherwise unspecified.

The `getBean` method returns the bean instance. The `getMethod` method returns the method of the bean class for which the interceptor was invoked. For `AroundInvoke` methods, this is the business method on the bean class; for lifecycle callback interceptor methods, `getMethod` returns null.

The `proceed` method causes the invocation of the next interceptor method in the chain, or, when called from the last `AroundInvoke` interceptor method, the business method. The `proceed` method returns the result of that method invocation. If the business method returns `void`, proceed returns `null`. For lifecycle callback interceptor methods, if there is no callback method defined on the bean class, the invocation of `proceed` in the last interceptor method in the chain is a no-op, and `null` is returned.

3.4.4 Exceptions

`AroundInvoke` methods run in the same Java call stack as the bean business method. `InvocationContext.proceed()` will throw the same exception as any thrown in the business method unless an interceptor further down the Java call stack has caught it and thrown a different exception. `AroundInvoke` developers should use try/catch/finally blocks around the `proceed()` method to handle any initialization and/or cleanup operations they want to invoke.

`AroundInvoke` methods are allowed to catch and suppress business method exceptions. `AroundInvoke` methods are allowed to throw runtime exceptions or any checked exceptions that the business method allows within its throws clause. If an `AroundInvoke` method throws an exception before it calls the `proceed()` method, no other `AroundInvoke` methods will be called. Since previous `AroundInvoke` methods are invoked in the same Java call stack, those methods may handle these exceptions in catch/finally blocks around the `proceed()` method call.

`AroundInvoke` methods can mark the transaction for rollback by throwing a runtime exception or by calling the EJBContext `setRollbackOnly()` method. `AroundInvoke` methods may cause this rollback before or after `InvocationContext.proceed()` is called.

3.5 Home Interfaces

The requirement for Home interfaces has been eliminated.

Session beans are no longer required to have home interfaces. A client may acquire a reference to a session bean through one of the mechanisms described in Chapter 8.

EJB 3.0 entities do not have home interfaces. A client may create an instance of an entity type by means of the `new` operation. The entity instance may be persisted by means of the `EntityManager` APIs defined in the "*Java Persistence*" document [2].

Chapter 4 Stateless Session Beans

This chapter describes requirements that are specific to stateless session beans.

4.1 Requirements for Stateless Session Beans

4.1.1 Business Interface

The business interface of a session bean written to the EJB 3.0 API is a plain Java interface, not an EJBObject or EJBLocalObject interface.

In the case of a session bean that implements a web service, a web service interface is not required to be defined. The `WebMethod` annotations are used to identify the methods that are exposed as web service operations. The session bean that serves as a web service endpoint is annotated with the `WebService` annotation. These annotations for web services are defined by JSR-181 [9].

4.1.2 Home Interface

Stateless session beans do not need home interfaces. The client may acquire a reference to a stateless session bean by means of one of the mechanisms described in Chapter 8.

4.1.3 Bean Class

A stateless session bean must be annotated with the `Stateless` annotation or denoted in the deployment descriptor as a stateless session bean. The bean class need not implement the `javax.ejb.SessionBean` interface.

4.1.4 Lifecycle Callbacks for Stateless Session Beans

The following lifecycle event callbacks are supported for stateless session beans[4]:

- `PostConstruct`
- `PreDestroy`

`PostConstruct` callbacks occur after any dependency injection has been performed by the container and before the first business method invocation on the bean.

`PostConstruct` methods are invoked in an unspecified transaction context and security context.

`PreDestroy` callbacks occur at the time the bean instance is destroyed.

`PreDestroy` methods execute in an unspecified transaction and security context.

4.1.5 Dependency Injection

If a stateless session bean uses dependency injection mechanisms for the acquisition of references to resources or other objects in its environment (see Chapter 8), the container injects these references before any business methods or lifecycle callback interceptor methods are invoked on the bean instance.

4.1.6 Interceptors for Stateless Session Beans

The `AroundInvoke` methods are supported for stateless session business method invocations. These interceptor methods may be defined on the bean class or on a interceptor class and apply to the invocation of the business methods of the bean. See Section 3.4 “Interceptors” .

[4] PostActivate and PrePassivate callbacks, if specified, are ignored for stateless session beans.

4.1.6.1 Example

```
@Stateless
@Interceptors({
     com.acme.AccountAudit.class,
     com.acme.Metrics.class,
     com.acme.CustomSecurity.class
})
public class AccountManagementBean implements AccountManagement {
    public void createAccount(int accountNumber, AccountDetails
details) { ... }
    public void deleteAccount(int accountNumber) { ... }
    public void activateAccount(int accountNumber) { ... }
    public void deactivateAccount(int accountNumber) { ... }
    ...
}

public class Metrics {
    @AroundInvoke
    public Object profile(InvocationContext inv) throws Exception {
       long time = System.currentTimeMillis();
       try {
          return inv.proceed();
       } finally {
         long endTime = time - System.currentTimeMillis();
         System.out.println(inv.getMethod() + " took " + endTime + "
milliseconds.");
       }
    }
}


public class AccountAudit {
    @AroundInvoke
    public Object auditAccountOperation(InvocationContext inv) throws
Exception {
       try {
         Object result = inv.proceed();
         Auditor.audit(inv.getMethod().getName(), inv.getParame-
ters[0]);
         return result;
       } catch (Exception ex) {
          Auditor.auditFailure(ex);
          throw ex;
       }
    }
}

public class CustomSecurity {
    @AroundInvoke
    public Object customSecurity(InvocationContext inv) throws Excep-
tion {
       doCustomSecurityCheck(inv.getEJBContext().getCallerPrinci-
pal());
       return inv.proceed();
    }
    private void doCustomSecurityCheck(Principal caller) throws
SecurityException {...}
}
```

4.2 Client View

The local or remote client of a session bean acquires a reference to a session bean business interface through one of the dependency injection or lookup mechanisms described in Chapter 8.

4.3 Other Requirements

The full set of requirements that apply to stateless session beans are specified in *"EJB Core Contracts and Requirements" [1]*.

Chapter 5 Stateful Session Beans

This chapter covers requirements that are specific to stateful session beans.

5.1 Requirements for Stateful Session Beans

5.1.1 Business Interface

The business interface of a session bean written to the EJB 3.0 API is a plain Java interface, not an EJBObject or EJBLocalObject interface.

5.1.2 Home Interface

Stateful session beans do not need home interfaces. The client may acquire a reference to a stateless session bean by means of one of the mechanisms described in Chapter 8.

5.1.3 Bean Class

A stateful session bean must be annotated with the `Stateful` annotation or denoted in the deployment descriptor as a stateful session bean. The bean class need not implement the `javax.ejb.Session-`

`Bean` interface or the `java.io.Serializable` interface.[5]

A stateful session bean may implement the `SessionSynchronization` interface, as described in "*EJB Core Contracts and Requirements*", Chapter 4 [1].

5.1.4 Lifecycle Callbacks for Stateful Session Beans

Stateful session beans support callbacks for the following lifecycle events: construction, destruction, activation, and passivation.

The lifecycle event callbacks are the following. They may be defined on the bean class or an interceptor class for the bean.[6]

- `PostConstruct`
- `PreDestroy`
- `PostActivate`
- `PrePassivate`

5.1.4.1 Semantics of the Life Cycle Callback Methods for Stateful Session Beans

`PostConstruct` methods are invoked on the newly constructed instance, after any dependency injection has been performed by the container and before the first business method is invoked on the bean.

`PostConstruct` methods are invoked in an unspecified transaction and security context.

`PreDestroy` methods execute after any method annotated with the `Remove` annotation has completed.

`PreDestroy` methods are invoked in an unspecified transaction and security context.

The semantics of `PrePassivate` and `PostActivate` are the same as the EJB 2.1 `ejbActivate` and `ejbPassivate` callback methods. See Chapter 4 of the "*EJB Core Contracts and Requirements*" document [1].

5.1.5 Dependency Injection

If a stateful session bean uses dependency injection mechanisms for the acquisition of references to resources or other objects in its environment (see Chapter 8), the container injects these references before any business methods or lifecycle callback interceptor methods are invoked on the bean instance.

---

[5] The container must be able to handle the passivation of the bean instance even if the bean class does not implement the Serializable interface. See the document "*EJB Core Contracts and Requirements*" [1], Chapter 4.

[6] The callbacks PreConstruct, PostDestroy, PreActivate, and PostPassivate were not introduced because there did not seem to be use cases that justified their introduction.

5.1.6 Interceptors for Stateful Session Beans

`AroundInvoke` methods are supported for stateful session business method invocations. These interceptor methods may be defined on the bean class or on a interceptor class and apply to the invocation of the business methods of the bean.

For stateful session beans that implement the `SessionSynchronization` interface, `afterBegin` occurs before any `AroundInvoke` method invocation, and `beforeCompletion` after all `AroundInvoke` invocations are finished.

5.1.7 Example

```
@Stateful
public class AccountManagementBean implements AccountManagement {
    @Resource SessionContext sessionContext;
    Socket cs;

    @PostConstruct
    @PostActivate
    public void initRemoteConnectionToAccountSystem() {
        try {
            cs = new Socket(DEST_HOST, DEST_PORT);
        } catch (Exception ex) {
            throw new EJBException("Could not allocate socket", ex);
        }
    }

    @PreDestroy
    @PrePassivate
    public void closeRemoteConnectionToAccountSystem() {
        try {
            cs.close();
        } catch (IOException ioEx) { // Ignore }
        cs = null;
    }

    public OpResult createAccount(int accountNumber, AccountDetails
details) { ... }
    public OpResult deleteAccount(int accountNumber) { ... }
    public OpResult activateAccount(int accountNumber) { ... }
    public OpResult deactivateAccount(int accountNumber) { ... }

    @Remove
    public void logOff() { ... }

    @AroundInvoke
    public Object auditAccountOperation(InvocationContext inv) throws
Exception {
          try {
              Object result = inv.proceed();
              if ((OpResult)result == OpResult.SUCCESS) {
                if (inv.getParameters[0].length > 0) {
                   Auditor.audit(inv.getMethod().getName(),
inv.getParameters[0], ..userInfo.. etc.);
                }
              }
              return result;
```

```
            } catch (Exception ex) {
            Auditor.auditFailure(inv.getMethod(), inv.getParameters(),
ex);
              throw ex;
            }
    }
}
```

5.1.8 Client View

The local or remote client of a session bean acquires a reference to the session bean business interface through one of the dependency injection or lookup mechanisms described in Chapter 8.

When a stateful session bean is looked up or otherwise obtained through the explicit JNDI lookup mechanisms, the container must provide a new stateful session bean instance, as required by the Java Platform, Enterprise Edition specification (Section “Java Naming and Directory Interface (JNDI) Naming Context” [7]).

When stateful session bean is injected into a client context or is obtained by lookup, the container creates a new stateful session bean instance to which method invocations from the client are delegated. This instance, however, is uninitialized from the client’s point of view, since as the client does not call an explicit “create” method to obtain and initialize the bean.

The client typically initializes a stateful session bean through business methods defined as part of by the bean’s interface. The bean may provide one or more initialization methods for this purpose.

5.1.9 Stateful Session Bean Removal

The `Remove` annotation may be used to annotate a stateful session bean business method. Use of this annotation will cause the container to remove the stateful session bean instance after the completion (normal or abnormal) of the annotated method.

5.1.9.1 Example

```
@Stateful public class ShoppingCartBean implements ShoppingCart {

    ...
    private String customer;

    public void startToShop(String customer) {
        this.customer = customer;
    ...
    }

    public void addToCart(Item item) {
        ...
    }

    @Remove public void finishShopping() {
        ...
    }
}
```

5.2 Other Requirements

The full set of requirements that apply to stateful session beans are specified in "*EJB Core Contracts and Requirements*" *[1]*.

Chapter 6 Message-Driven Beans

This chapter describes requirements that are specific to message-driven beans.

6.1 Requirements for Message-Driven Beans

6.1.1 Business Interface

The business interface of a message-driven bean is the message-listener interface that is determined by the messaging type in use for the bean. For example, in the case of JMS, this is the `javax.jms.MessageListener` interface.

The message-driven bean must implement the appropriate message listener interface for the messaging type that the message-driven bean supports or must designate its message listener interface using the `MessageDriven` annotation or the deployment descriptor.

6.1.2 Bean Class

A message driven bean must be annotated with the `MessageDriven` annotation or denoted in the deployment descriptor as a message-driven bean. The bean class need not implement the `javax.ejb.MessageDrivenBean` interface.

6.1.3 Lifecycle Callbacks for Message-Driven Beans

The following lifecycle event callbacks are supported for message-driven beans:

- `PostConstruct`
- `PreDestroy`

`PostConstruct` callbacks occur before the first message listener method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

`PostConstruct` callback methods execute in an unspecified transaction and security context.

`PreDestroy` callbacks occur at the time the bean is removed from the pool or destroyed.

`PreDestroy` callback methods execute in an unspecified transaction and security context.

6.1.4 Dependency Injection

If a message-driven bean uses dependency injection mechanisms for the acquisition of references to resources or other objects in its environment (see Chapter 8), the container injects these references before any business methods or lifecycle callback interceptor methods are invoked on the bean instance.

6.1.5 Interceptors for Message-Driven Beans.

The `AroundInvoke` methods are supported for message-driven beans. These interceptor methods may be defined on the bean class or on a interceptor class and apply to the handling of the bean's message listener method invocation.

6.2 Other Requirements

The full set of requirements that apply to message-driven beans are specified in "*EJB Core Contracts and Requirements*" *[1]*.

Chapter 7 Entity Beans and Persistence

The model for entity beans, persistence, and object/relational mapping has been considerably revised and enhanced in the Enterprise JavaBeans 3.0 release.

An EJB 3.0 entity is a lightweight persistent domain object.

The contracts and requirements for entities defined by Enterprise JavaBeans 3.0 are specified in the document "*Java Persistence*" [2], which also contains the full specification of the Enterprise JavaBeans Query Language (EJB QL), and the metadata for object/relational mapping.

Chapter 8 Enterprise Bean Context and Environment

The enterprise bean's context comprises its container context and its resource and environment context.

The bean may gain access to references to resources and other environment entries in its context by having the container supply it with those references. In this case, bean instance variables or setter methods are annotated as target for dependency injection.

Alternatively, the `lookup` method added to the `javax.ejb.EJBContext` interface or the JNDI APIs may be used to look up resources in the bean's environment. (See Section 8.1.4.)

The same set of metadata annotations are used to express context dependencies for both these approaches.

8.1 Annotation of Context Dependencies

A bean declares a dependency upon a resource or other entry in its environment context through a dependency annotation.

A dependency annotation specifies the type of object or resource on which the bean is dependent, its characteristics, and the name through which it is to be accessed.

The following are examples of dependency annotations:

```
@EJB(name="mySessionBean", beanInterface=MySessionIF.class)

@Resource(name="myDB", type=javax.sql.DataSource.class)
```

Dependency annotations may be attached to the bean class or to its instance variables or methods.

The amount of information that needs to be specified for a dependency annotation depends upon its usage context and how much information can be inferred from that context. See Chapter 15 (Enterprise Bean Environment) in *"EJB Core Contracts and Requirements" [1]*.

The following sections discuss and illustrate the various approaches.

8.1.1 Annotation of Instance Variables

The developer may annotate instance variables of the enterprise bean class to indicate dependencies upon resources or other objects in the bean's environment. The container automatically initializes these annotated variables with the external references to the specified environment objects. This initialization occurs before any business methods are invoked on the bean instance and after the time the the bean's EJBContext is set.

Example:

```
@Stateless public class MySessionBean implements MySession {

    @Resource(name="myDB") //type is inferred from variable
    public DataSource customerDB;

    @EJB //reference name and type inferred from variable
    public AddressHome addressHome;

 public void myMethod1(String myString)
  {
    try
     {
       Connection conn = customerDB.getConnection();
       ...
     }
     catch (Exception ex)
     {
       ...
     }
   }

  public void myMethod2(String myString)
  {
   Address a = addressHome.create(myString);
  }
}
```

When the resource type can be determined by the variable type, the annotation need not contain the type of the object to be accessed. If the name for the resource reference in the bean's environment is the same

as the variable name, it does not need to be explicitly specified. See Chapter 15, *"Enterprise Bean Environment,"* in the document *"EJB Core Contracts and Requirements"* [1].

Examples

```
@EJB public ShoppingCart myShoppingCart;

@Resource public DataSource myDB;

@Resource public UserTransaction utx;

@Resource SessionContext ctx;
```

8.1.2 Setter Injection

Setter injection provides an alternative to the container's initialization of variables described above.

When setter injection is to be used, the dependency annotations are applied to setter methods of the bean class defined for that purpose.

Examples

```
@Resource(name="customerDB")
public void setDataSource(DataSource myDB) {
    this.ds = myDB;
}


@Resource // reference name is inferred from the property name
public void setCustomerDB(DataSource myDB) {
        this.customerDB = myDB;
}

@Resource
public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
}
```

When the resource type can be determined by the parameter type, the annotation need not specify the type of the object to be accessed. If the name of the resource is the same as the property name corresponding to the setter method, it does not need to be explicitly specified.

A setter method that is annotated with the `Resource` or other dependency annotation will be used by the container for dependency injection. Such setter injection methods will be called by the container before any business methods are invoked on the bean instance and after the bean's EJBContext is set.

8.1.3 Injection and Lookup

Resources, references to components, and other objects that may be looked up in the JNDI name space may be injected by means of the injection mechanisms listed above.

References to injected objects are looked up name. These lookups are performed in the referencing

bean's `java:comp/env` namespace as specified in *EJB Core Contracts and Requirements*" [1], Chapter 15, "Enterprise Bean Environment."

8.1.4 EJBContext

The method `Object lookup(String name)` is added to the `javax.ejb.EJBContext` interface. This method can be used to lookup resources and other environment entries bound in the bean's JNDI environment naming context.

Injection of the bean's EJBContext object may be obtained as described in sections 8.1.1 and 8.1.2 above.

Chapter 9 Compatibility and Migration

This chapter addresses issues of compatibility and migration between EJB 3.0 and earlier components and clients.

9.1 Support for Existing Applications

Existing EJB 2.1 and earlier applications must be supported to run unchanged in EJB 3.0 containers. All EJB 3.0 implementations must support EJB 1.1, EJB 2.0, and EJB 2.1 deployment descriptors for applications written to earlier versions of the Enterprise JavaBeans specification.

9.2 Interoperability of EJB 3.0 and Earlier Components

This release of Enterprise JavaBeans supports migration and interoperability among client and server components written to different versions of the EJB APIs as described below.

9.2.1 Clients written to the EJB 2.x APIs

An enterprise bean that is written to the EJB 2.1 or earlier API release may be a client of components written to EJB 3.0 API using the earlier EJB APIs when deployed in an EJB 3.0 container.

Such an EJB 2.1 or earlier client component does not need to be rewritten or recompiled to be a client of a component written to the EJB 3.0 API.

Such clients may access components written to the EJB 3.0 APIs and components written to the earlier EJB APIs within the same transaction.

See Section 9.3 for a discussion of the mechanisms that are used to enable components written to the EJB 3.0 API to be accessed and utilized by clients written to earlier versions of the EJB specification.

9.2.2 Clients written to the new EJB 3.0 API

A client written to the EJB 3.0 API may be a client of a component written to the EJB 2.1 or earlier API.

Such clients may access components written to the EJB 3.0 APIs and components written to the earlier EJB APIs within the same transaction.

Such clients access components written to the earlier EJB APIs using the EJB 2.1 client view home and component interfaces. The `EJB` annotation may be used for the injection of home interfaces into components that are clients of beans written to the earlier EJB client view. See Section 10.10.

9.2.3 Combined use of EJB 2.x and EJB 3.0 persistence APIs

EJB clients may access EJB 3.0 entities and/or the EntityManager together with EJB 2.x entity beans together within the same transaction as well as within separate transactions.[7]

9.2.4 Other Combinations of EJB 3.0 and Earlier APIs

The "*EJB Core Contracts and Requirements*" document [1] specifies how the new EJB 3.0 APIs may be used together with the existing EJB APIs defined in [3] within a single component class. Such usage may be helpful in facilitating incremental migration of existing applications to EJB 3.0.

9.3 Adapting EJB 3.0 Session Beans to Earlier Client Views

Clients written to the EJB 2.1 and earlier client view depend upon the existence of a home and component interface.

A session bean written to the EJB 3.0 API may be adapted to such earlier preexisting client view interfaces.

The session bean designates the interfaces to be adapted by using the `RemoteHome` and/or `LocalHome` metadata annotations (or equivalent deployment descriptor elements).

When the client is deployed, the container classes that implement the EJB 2.1 home and remote inter-

[7] In general, the same database data should not be accessed by both EJB 3.0 and EJB 2.x entities within the same application: behavior is unspecified if data aliasing occurs.

faces (or local home and local interfaces) referenced by the client must provide the implementation of the `javax.ejb.EJBHome` and `javax.ejb.EJBObject` interfaces (or the `javax.ejb.EJBLocalHome` and `javax.ejb.EJBLocalObject` interfaces) respectively.

In addition, the container implementation classes must implement the methods of the home and component interfaces to apply to the EJB 3.0 component being referenced as described below.

9.3.1 Stateless Session Beans

The invocation of the home `create()` method must return the corresponding local or remote component interface of the bean. This may or may not cause the creation of the bean instance, depending on the container's implementation strategy. For example, the container may preallocate bean instances (e.g., in a pooling strategy) or may defer the creation of the bean instance until the first invocation of a business method on the bean class. When the bean instance is created, the container invokes the `setSessionContext` method (if any), performs any other dependency injection, and invokes the `PostConstruct` lifecycle callback method(s) (if any), as specified in "*EJB Core Contracts and Requirements*" [1].

It is likewise implementation-dependent as to whether the invocation of the EJBHome `remove(Handle)` or EJBObject or EJBLocalObject `remove()` method causes the immediate removal of the bean instance, depending on the container's implementation strategy. When the bean instance is removed, the `PreDestroy` callback method (if any) is invoked, as specified in Section 4.1.4.

The invocations of the business methods of the component interface are delegated to the bean class.

9.3.2 Stateful Session Beans

The invocation of a `create<METHOD>()` method causes construction of the bean instance, invocation of the `PostConstruct` callback (if any), and invocation of the matching `Init` method, and returns the corresponding local or remote component interface of the bean. The invocation of these methods occurs in the same transaction and security context as the client's call to the `create` method.

The invocation of the EJBHome `remove(Handle)` or EJBObject or EJBLocalObject `remove()` method causes the invocation of the the `PreDestroy` callback method (if any) and removal of bean instance, as described in "*EJB Core Contracts and Requirements*" [1].

The invocations of the business methods of the component interface are delegated to the bean class.

The `Init` annotation is used to specify the correspondence of a method on the bean class with a create method of the adapted EJBHome and/or EJBLocalHome interface. The result type of such an `Init` method is required to be `void`, and its parameter types must be exactly the same as those of the referenced create method.

9.4 Combined Use of EJB 3.0 and EJB 2.1 APIs in a Bean Class

This document describes the typical usage of annotations to specify the enterprise bean type and callback methods. It is permitted to combine the use of such annotations with the bean's implementation of one of the `javax.ejb.EnterpriseBean` interfaces as such combination may be useful in facilitating migration to the EJB 3.0 simplified programming model.

In addition to the business interface described in Section 3.2, a session bean may define EJBHome, EJBLocalHome, EJBObject, and/or EJBLocalObject interfaces in accordance with the rules of the EJB 2.1 specification. A deployment descriptor or metadata annotations may be used to associate the bean class with these interfaces.

Requirements for the combined usage of EJB 3.0 and EJB 2.1 and earlier APIs within an enterprise bean class are defined in the specification document "*EJB Core Contracts and Requirements*."

Chapter 10 Metadata Annotations

This chapter defines the metadata annotations introduced by this specification.

These annotations are in the `javax.ejb` package except where otherwise indicated.

Annotations related to persistence are defined in the document "*Java Persistence*" [2].

10.1 Annotations to Specify Bean Type

10.1.1 Stateless Session Beans

The `Stateless` annotation specifies that the enterprise bean is a stateless session bean. The `Stateless` annotation is applied to the bean class.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Stateless {
     String name() default "";
     String mappedName() default "";
     String description() default "";
}
```

The `name` annotation element defaults to the unqualified name of the bean class.

The `mappedName` element is a product-specific name that the session bean should be mapped to. Applications that use mapped names may not be portable.

10.1.2 Stateful Session Beans

The `Stateful` annotation specifies that the enterprise bean is a stateful session bean. The `Stateful` annotation is applied to the bean class.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Stateful {
     String name() default "";
     String mappedName() default "";
     String description() default "";
}
```

The `name` annotation element defaults to the unqualified name of the bean class.

The `mappedName` element is a product-specific name that the session bean should be mapped to. Applications that use mapped names may not be portable.

10.1.2.1 Init Annotation for Stateful Session Beans

The `Init` annotation is used to specify the correspondence of a method on the bean class with a `create<METHOD>` method for an adapted EJB 2.1 EJBHome and/or EJBLocalHome client view. The result type of such an `Init` method is required to be `void`, and its parameter types must be exactly the same as those of the referenced `create<METHOD>` method(s).

```
@Target(METHOD) @Retention(RUNTIME)
public @interface Init{
     String value() default {};
}
```

The `value` element must be specified when the `Init` annotation is used in association with an adapted home interface of a stateful session bean that has more than one `create<METHOD>` method. It specifies the name of the corresponding `create<METHOD>` method of the adapted home.

The `Init` method is only required to be specified for stateful session beans that provide a `RemoteHome` or `LocalHome` interface. The name of the adapted create method of the Home or LocalHome interface must be specified if there is any ambiguity.

10.1.2.2 Remove Annotation for Stateful Session Beans

The `Remove` annotation is used to denote a remove method of a stateful session bean. Completion of this method causes the container to destroy the stateful session bean, first invoking the bean's `PreDestroy` method, if any. The `retainIfException` element allows the removal to be prevented if the method terminates abnormally with an application exception.

```
@Target(METHOD) @Retention(RUNTIME)
```

```
public @interface Remove{
     boolean retainIfException() default false;
}
```

10.1.3 Message-driven Beans

The `MessageDriven` annotation specifies that the enterprise bean is a message-driven bean. This annotation is applied to the bean class.

The `name` annotation element defaults to the unqualified name of the bean class.

The `messageListenerInterface` element specifies the message listener interface of the bean. It must be specified if the bean class does not implement its message listener interface or implements more than one interface other than `java.io.Serializable`, `java.io.Externalizable`, or any of the interfaces defined by the `javax.ejb` package.

The `mappedName` element is a product-specific name that the message-driven bean should be mapped to. Applications that use mapped names may not be portable.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface MessageDriven {
     String name() default "";
     Class messageListenerInterface default Object.class;
     ActivationConfigProperty[] activationConfig() default {};
     String mappedName() default "";
     String description() default "";
}

public @interface ActivationConfigProperty {
  String propertyName();
  String propertyValue();
}
```

10.2 Annotations to Specify Local or Remote Interfaces

The `Remote` and `Local` annotations apply only to session beans and their interfaces.

The `Remote` annotation is applied to the session bean class or remote business interface to designate a remote interface of the bean.

The `Local` annotation is applied to the session bean class or local business interface to designate a local interface of the bean.

Use of the `Local` annotation is only required when the bean class does not implement only a single interface other than any of the following: `java.io.Serializable`; `java.io.Externalizable`; any of the interfaces defined in `javax.ejb`.

The `value` element is specified only when the annotation is applied to the bean class. It is only required to be specified if the bean class implements more than one interface (excluding

`java.io.Serializable`, `java.io.Externalizable`, and any of the interfaces defined by the `javax.ejb` package).

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Remote {
     Class[] value() default {}; // list of remote business interfaces
}

@Target(TYPE) @Retention(RUNTIME)
public @interface Local {
     Class[] value() default {}; // list of local business interfaces
}
```

10.3 Annotations to Support EJB 2.1 and Earlier Client View

The `RemoteHome` and `LocalHome` annotations may be applied to session beans only.

These annotations are intended for use with EJB 3.0 session beans that provide an adapted EJB 2.1 component view. They may also be used with beans that have been written to the EJB 2.1 APIs.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface RemoteHome {
  Class value();   // home interface
}
@Target(TYPE) @Retention(RUNTIME)
public @interface LocalHome {
  Class value();   // local home interface
}
```

10.4 TransactionManagement

The `TransactionManagement` annotation specifies the transaction management demarcation type of a session bean or message-driven bean. If the `TransactionManagement` annotation is not specified for a session bean or message-driven bean, the bean is assumed to have container managed transaction demarcation.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface TransactionManagement {
     TransactionManagementType value()
               default TransactionManagementType.CONTAINER;
}
```

The enum `TransactionManagementType` is used to specify whether container-managed or bean-managed transaction management is used.

```
public enum TransactionManagementType {
  CONTAINER,
  BEAN
}
```

10.5 Transaction Attributes

The `TransactionAttribute` annotation specifies whether the container is to invoke a business method within a transaction context. The semantics of transaction attributes are defined in Chapter 12 of the "*EJB Core Contracts and Requirements*" document [1].

The `TransactionAttribute` annotation can only be specified if container managed transaction demarcation is used. The annotation can be specified on the bean class and/or it can be specified on methods of the class that are methods of the business interface. Specifying the `TransactionAttribute` annotation on the bean class means that it applies to all applicable business interface methods of the class. Specifying the annotation on a method applies it to that method only. If the annotation is applied at both the class and the method level, the method value overrides if the two disagree.

The values of the `TransactionAttribute` annotation are defined by the enum `TransactionAttributeType`.

If a `TransactionAttribute` annotation is not specified, and the bean uses container managed transaction demarcation, the semantics of the `REQUIRED` transaction attribute are assumed.

```
public enum TransactionAttributeType {
   MANDATORY,
   REQUIRED,
   REQUIRES_NEW,
   SUPPORTS,
   NOT_SUPPORTED,
   NEVER
}

@Target({METHOD, TYPE}) @Retention(RUNTIME)
public @interface TransactionAttribute {
  TransactionAttributeType value()
         default TransactionAttributeType.REQUIRED;
}
```

10.6 Interceptors and LifeCycle Callbacks

The `Interceptors` annotation is used to designate one or more interceptor classes associated with a bean. The `Interceptors` annotation is applied to the bean class or to a business method of the bean.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Interceptors {
     Class[] value();
}
```

The `AroundInvoke` annotation is used to designate an interceptor method.

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface AroundInvoke {}
```

The `ExcludeDefaultInterceptors` annotation, when applied to a bean class, excludes the invocation of default interceptors for all business methods of the bean. When applied to a business method, it excludes the invocation of default interceptors for that method.

```
@Target({TYPE, METHOD}) @Retention(RUNTIME)
public @interface ExcludeDefaultInterceptors {}
```

The `ExcludeClassInterceptors` annotation excludes the invocation of class-level interceptors (but not default interceptors) for the given method.

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface ExcludeClassInterceptors {}
```

The `javax.annotation.PostConstruct`, `javax.annotation.PreDestroy`, and the `javax.ejb.PostActivate and javax.ejb.PrePassivate` annotations designate lifecycle callback methods.

```
package javax.annotation;
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostConstruct {}

package javax.annotation;
@Target({METHOD}) @Retention(RUNTIME)
public @interface PreDestroy {}

package javax.ejb;
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostActivate {}

package javax.ejb;
@Target({METHOD}) @Retention(RUNTIME)
public @interface PrePassivate {}
```

The `javax.annotation.PostConstruct` and `javax.annotation.PreDestroy` annotations are defined in the specification "*Common Annotations for the Java Platform*" [8].

10.7 Timeout

The `Timeout` annotation is used to denote the timeout method of an enterprise bean.

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface Timeout {}
```

10.8 Exceptions

The `ApplicationException` annotation is applied to an exception to denote that it is an application exception and should be reported to the client directly (i.e., unwrapped). The `ApplicationException` annotation may be applied to both checked and unchecked exceptions. The `rollback` element is used to indicate whether the container must cause the transaction to rollback when the excep-

tion is thrown.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface ApplicationException {
    boolean rollback() default false;
}
```

10.9 Security and Method Permissions

The following security-related annotations are in the package `javax.annotation.security`. They are defined by [8], and are presented here for reference.

10.9.1 Security Role References

The `DeclareRoles` annotation is used to declare the references to security roles in the enterprise bean code.

```
package javax.annotation.security;
@Target({TYPE}) @Retention(RUNTIME)
public @interface DeclareRoles {
    String[] value();
}
```

10.9.2 MethodPermissions

The `RolesAllowed` annotation specifies the security roles that are allowed to invoke the methods of the bean. The value of the `RolesAllowed` annotation is a list of security role names.

This annotation can be specified on the bean class and/or it can be specified on methods of the class that are methods of the business interface. Specifying the `RolesAllowed` annotation on the bean class means that it applies to all applicable interface methods of the class. Specifying the annotation on a method applies it to that method only. If the annotation is applied at both the class and the method level, the method value overrides if the two disagree. If the `PermitAll` annotation is applied to the bean class, and `RolesAllowed` is specified on an individual method, the value of the `RolesAllowed` annotation overrides for the given method.

```
@Target({TYPE, METHOD}) @Retention(RUNTIME)
public @interface RolesAllowed {
  String[] value();
  }
```

10.9.3 PermitAll

The `PermitAll` annotation specifies that all security roles are allowed to invoke the specified method(s)—i.e., that the specified method(s) are “unchecked”. This annotation can be specified on the bean class and/or it can be specified on the business methods of the class. Specifying the `PermitAll` annotation on the bean class means that it applies to all applicable business methods of the class. Specifying the annotation on a method applies it to that method only, overriding any class-level setting for the

particular method.

```
package javax.annotation.security;
@Target ({TYPE, METHOD}) @Retention(RUNTIME)
public @interface PermitAll {}
```

10.9.4 DenyAll

The `DenyAll` annotation specifies that no security roles are allowed to invoke the specified method—i.e. that the specified method is to be excluded from execution.

```
package javax.annotation.security;
@Target (METHOD) @Retention(RUNTIME)
public @interface DenyAll {}
```

10.9.5 RunAs

The `RunAs` annotation is used to specify the bean's run-as property. This annotation is applied to the bean class. Its value is the name of a security role.

```
package javax.annotation.security;
@Target(TYPE) @Retention(RUNTIME)
public @interface RunAs {
   String value();
  }
```

10.10 EJB References

The `EJB` annotation denotes a reference to an EJB business interface or home interface.

The `name` element refers to the name by which the resource is to be looked up in the environment. The `beanInterface` element is the referenced interface type—either the business interface or home interface.

The `beanName` element references the value of the `name` element of the `Stateful` or `Stateless` annotation (or `ejb-name` element, if the deployment descriptor was used to define the name of the bean). The `beanName` element allows disambiguation if multiple session beans in the ejb-jar implement the same interface.

The `mappedName` element is a product-specific name that the bean reference should be mapped to. Applications that use mapped names may not be portable.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface EJB {
    String name() default "";
    Class beanInterface() default Object.class;
    String beanName() default "";
    String mappedName() default "";
```

```
    String description() default "";
}

@Target(TYPE) @Retention(RUNTIME)
public @interface EJBs {
  EJB[] value();
}
```

10.11 Resource References

The `Resource` and `Resources` annotations are in the package `javax.annotation`. They are defined by [8], and are presented here for reference.

The `Resource` annotation is used to express a dependency on an external resource in the bean's environment. The `name` property refers to the name by which the resource is to be known in the environment; the type is the resource manager connection factory type. The `authenticationType` element specifies whether the container or bean is to perform authentication. The `shareable` element refers to the sharability of resource manager connections. The `mappedName` element is a product-specific name that the resource should be mapped to. Applications that use mapped names may not be portable.

```
package javax.annotation;
@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 "";
}


package javax.annotation;
@Target(TYPE) @Retention(RUNTIME)
public @interface Resources {
  Resource[] value();
}
```

Chapter 11 Related Documents

[1] Enterprise JavaBeans, v 3.0. EJB Core Contracts and Requirements.

[2] Enterprise JavaBeans, v 3.0. Java Persistence. *http://java.sun.com/products/ejb.*

[3] Enterprise JavaBeans, v 2.1. *http://java.sun.com/products/ejb.*

[4] Java Naming and Directory Interface (JNDI). *http://java.sun.com/products/jndi*.

[5] Java Remote Method Invocation (RMI). *http://java.sun.com/products/rmi.*

[6] Java Transaction API (JTA). *http://java.sun.com/products/jta*.

[7] Java Platform, Enterprise Edition (Java EE), v 5. *http://java.sun.com/javaee*.

[8] JSR-250: Common Annotations for the Java Platform. *http://jcp.org/en/jsr/detail?id=250*.

[9] JSR-181: Web Services Metadata for the Java Platform. *http//jcp.org/en/jsr/detail?id=181*.

[10] JSR-175: A Metadata Facility for the Java Programming Language. *http://jcp.org/en/jsr/detail?id=175*.

[11] Web Services for Java EE, v 1.2.

[12] Java Message Service (JMS), v 1.1. *http://java.sun.com/products/jms*.

Appendix A Revision History

This appendix lists the significant changes that have been made during the development of the EJB 3.0 specification.

A.1 Early Draft 1

Created document.

A.2 Early Draft 2

Split Early Draft 1 document into two documents, this document and "*Persistence API*" [2].

Added Overview chapter.

Moved discussion of items related to combined use of EJB 3.0 annotations and other new features with EJB 2.1 style components to separate chapter.

Added support for annotated callbacks and callback listener classes.

Added support for interceptors for session beans and message-driven beans.

Removed UniversalContext.

Added annotations for interceptors and callbacks.

Added chapter specifying the required support for the interoperability of components written to different versions of the EJB specification.

Added clarifications about relationships between metadata annotations and the deployment descriptor.

Separated out TransactionManagementType from Stateless, Stateful, and MessageDriven annotations as a separate annotation.

Renamed REQUIRESNEW as REQUIRES_NEW, NOTSUPPORTED as NOT_SUPPORTED.

Added Related Documents section.

Updated numerous examples.

A.3 Public Draft

Added ApplicationException annotation.

Clarified meaning of interceptor proceed() method.

Removed requirements for support of generated interfaces. Generation of interfaces may be supported by tools.

Added annotations for specification of local and remote interfaces.

Clarified that stateful session beans are not required to implement Serializable.

Updates to security and resource annotations.

Added support for dependency injection for interceptor classes.

Miscellaneous updates to reflect "*EJB Core Contracts and Requirements*" document.

A.4 Proposed Final Draft

Removed Interceptor annotation, since only Interceptors is needed.

Added support for method-level interceptors and default interceptors.

Merged lifecycle callbacks with interceptors.

Updated to reflect changes in JSR 250.