Contexts and Dependency Injection for the Java EE platform

CDI 1.1 Expert Group

Specification lead

Pete Muir, Red Hat Middleware, LLC

Version

CDI 1.1 Public Review Draft

26 October 2012

Table of Contents

Evaluation license

Copyright 2009 Red Hat Middleware LLC.

All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Red Hat Middleware LLC and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Red Hat Middleware LLC hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Red Hat Middleware LLC's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (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. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) 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;

(ii) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and

(iii) includes the following notice:

"This is an implementation of an early-draft specification developed under the Java Community Process (JCP). The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the Specification is contingent upon your agreement to terminate development and distribution of your implementation of early draft upon final completion of the Specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, para or interest in or to the Specification or any other Red Hat Middleware LLC intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Red Hat Middleware LLC if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.redhat" or their equivalents in any subsequent naming convention adopted through the Java Community Process, or any recognized successors or replacements thereof.

TRADEMARKS

No right, para, or interest in or to any trademarks, service marks, or trade names of Red Hat Middleware LLC or Red Hat'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.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY RED HAT MIDDLEWARE LLC. RED HAT MIDDLEWARE LLC MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

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. RED HAT MIDDLEWARE LLC 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 RED HAT MIDDLEWARE LLC 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 RED HAT MIDDLEWARE LLC AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Red Hat Middleware LLC (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 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

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 Red Hat Middleware LLC with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Red Hat Middleware LLC 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.

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.

Foreword

Contexts and Dependency Injection 1.1 (JSR-346) is an update to Contexts and Dependency Injection 1.0 (JSR-299). A full changelog can be found in the issue tracker release notes for CDI 1.1 [https://issues.jboss.org/secure/ReleaseNote.jspa?projectId=12311062&version=12315956].

1. Major changes

1.1. 1.1 Early Draft Review

- `@Disposes` methods for producer fields
- The `CDI` class, which provides programmatic access to CDI facilities from outside a managed bean
- Pass the qualifiers an event was fired with to the `ObserverMethod`
- Ability to veto beans declaratively using `@Veto` and `@Requires`
- Ability to access the `BeanManager` from the `ServletContext`
- Conversations in Servlet requests
- Application lifecycle events in Java EE
- Injection of `Bean` metadata into bean instances
- Programmatic access to a container provided `Producer`, `InjectionTarget`, `AnnotatedType`
- Ability to override attributes of a `Bean` via `BeanAttributes`
- Ability to process modules via `ProcessModule`
- Ability to wrap the `InjectionPoint`
- Ability to obtain `Extension` instances from `BeanManager`
- Injection of the `ServletContext`
- Access to `beans.xml` in `ProcessModule`
- Injection into enums

1.2. 1.1 Public Review Draft

- `ProcessAnnotatedType` fired for annotations
- Moved all Bean Validation integration to Bean Validation Specification as Bean Validation 1.1 PRD
- Clarify what beans are available during container lifecycle events
- Honor `WEB-INF/classes/META-INF/beans.xml` to activate `WEB-INF/classes` a bean archive
- Global ordering and enablement of interceptors and decorators
- Global selection of alternatives
- @New deprecated
- JMS MessageListener removed as not available in Java EE
- Support for unmanaged instances via `Instance.destroy()`

- Clarify interceptors and decorators must be implemented using proxying
- Allow multiple annotated types per Java class
- Allow Extensions to specify the annotations that they are interested in

2. Open issues

The expert group still has a number of topics under discussion, as of the Public Review Draft. Some of these are minor, alignment issues, or naming issues, however some of these directly affect core CDI concepts, and these are listed here.

2.1. Bean visibility

The CDI 1.0 specification clearly states that *only* beans whose bean class is accessible (using standard classloader visibility rules) can be injected into another bean. For example, if you have a bean A in WAR, assuming standard Java EE classloader structure, it wouldn't be available for injection in bean B, in an EJB modiule. This generally makes sense, as the type is not visible either.

CDI also offers two options to replace bean implementations transparently, without explicitly selecting that implementation (either by type or using a qualifier) - alternatives and specialization. In this case, it is less clear that the bean class of the specializing bean, or the bean class selected alternative, must be visible.

The CDI EG is still debating this issue, including whether to offer a backwards incompatible mode here.

2.2. `@ApplicationScoped` beans shared between all EAR modules

CDI implementations have not consistently shared `@ApplicationScoped` beans accross all modules of an EAR. This issue heavily relates to *Bean visibility*. The CDI 1.1 specification will clearly state how `@ApplicationScoped` are shared.

2.3. Startup event

A commonly requested feature is for the application to be able to do some work once the application has started but before it starts servicing requests originating remotely. Currently CDI 1.1 defines a `@Initialized(ApplicationScoped.class)` which is called when the application context starts, but we are investigating whether this can be extended to provide a more general startup event.

If we define such an event, we need to allow custom contexts to activate themselves whilst it is executing, however this is likely beyond the scope of CDI 1.1 and will likely be addressed in CDI 2.0.

2.4. `@WithAnnotations`

CDI 1.1 adds `@WithAnnotations` which allows an extension observing `ProcessAnnotatedType` to filter which types it sees. We would like to provide such functionality for all container lifecycle event observers, but there are some interesting things to consider, including whether it would be better to filter on qualifiers for later events. CDI 1.1 may or may not add such support, and we are looking for feedback on this.

2.5. Allowing arrays as qualifier members

CDI 1.0 requires array valued members of qualifiers to be annotated with `@Nonbinding`, excluding them from the resolution process. The JDK defines that annotation equality for array valued members should use `Arrays.equals()`, which requires two identical arrays (equal values, in the same order) in order to return true.

We feel that to make array valued members of qualifiers useful, we need to offer a pluggable strategy for checking equality of arrays, as often it would be desirable to consider two arrays with the same values, in any order, as equal. We intend to add this for CDI 1.1.

2.6. Restricting what CDI scans

CDI 1.0 will scan all classes in a jar with `beans.xml`. We plan to add a syntax to `beans.xml` that will the application developer to exclude classes using a variety of filtering options (e.g. by package). Weld offers such a syntax, and will be used as a starting point for CDI http://docs.jboss.org/weld/reference/1.1.5.Final/en-US/html/configure.html#d0e5769.

2.7. Observer resolution

CDI 1.0 requires the type used for observer resolution to be based on the runtime type of the event object. Unfortunately, the JDK erases generic type information about objects that we need to allow firing of many events with parameterized types. CDI 1.0 also completely ignores the generic type of the injected event object, which does typically contain the needed type information. We therefore intend to change the event observer resolution rules to allow the generic type of the event object to be taken into account if the runtime event object does not contain sufficient information.

Note that this may seem like a backwards incompatible change, however CDI 1.0 is essentially unimplementable today - examples in the spec do not work as described.

Chapter 1. Architecture

This specification defines a powerful set of complementary services that help improve the structure of application code.

- A well-defined lifecycle for stateful objects bound to *lifecycle contexts*, where the set of contexts is extensible
- A sophisticated, typesafe *dependency injection* mechanism, including the ability to select dependencies at either development or deployment time, without verbose configuration
- Support for Java EE modularity and the Java EE component architecture—the modular structure of a Java EE application is taken into account when resolving dependencies between Java EE components
- Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSF or JSP page
- The ability to *decorate* injected objects
- The ability to associate interceptors to objects via typesafe *interceptor bindings*
- An *event notification* model
- A web *conversation context* in addition to the three standard web contexts defined by the Java Servlets specification
- An SPI allowing *portable extensions* to integrate cleanly with the container

The services defined by this specification allow objects to be bound to lifecycle contexts, to be injected, to be associated with interceptors and decorators, and to interact in a loosely coupled fashion by firing and observing events. Various kinds of objects are injectable, including EJB 3 session beans, managed beans and Java EE resources. We refer to these objects in general terms as *beans* and to instances of beans that belong to contexts as *contextual instances*. Contextual instances may be injected into other objects by the dependency injection service.

To take advantage of these facilities, the developer provides additional bean-level metadata in the form of Java annotations and application-level metadata in the form of an XML descriptor.

The use of these services significantly simplifies the task of creating Java EE applications by integrating the Java EE web tier with Java EE enterprise services. In particular, EJB components may be used as JSF managed beans, thus integrating the programming models of EJB and JSF.

It's even possible to integrate with third-party frameworks. A portable extension may provide objects to be injected or obtain contextual instances using the dependency injection service. The framework may even raise and observe events using the event notification service.

An application that takes advantage of these services may be designed to execute in either the Java EE environment or the Java SE environment. If the application uses Java EE services such as transaction management and persistence in the Java SE environment, the services are usually restricted to, at most, the subset defined for embedded usage by the EJB specification.

1.1. Contracts

This specification defines the responsibilities of:

- the application developer who uses these services, and
- the vendor who implements the functionality defined by this specification and provides a runtime environment in which the application executes.

This runtime environment is called the *container*. For example, the container might be a Java EE container or an embeddable EJB container.

Chapter 2, *Concepts*, Chapter 3, *Programming model*, Chapter 4, *Inheritance and specialization*, Chapter 9, *Interceptor bindings*, Section 8.1, “Decorator beans” and Section 10.4, “Observer methods” define the programming model for Java EE components that take advantage of the services defined by this specification, the responsibilities of the component developer, and the annotations used by the component developer to specify metadata.

Chapter 5, *Dependency injection, lookup and EL*, Chapter 6, *Scopes and contexts*, Chapter 7, *Lifecycle of contextual instances*, Chapter 8, *Decorators*, Chapter 10, *Events* and Section 9.5, “Interceptor resolution” define the semantics and behavior of the services, the responsibilities of the container implementation and the APIs used by the application to interact directly with the container.

Chapter 12, *Packaging and deployment* defines how Java EE applications that use the services defined by this specification must be packaged into bean archives, and the responsibilities of the container implementation at application initialization time.

Chapter 11, *Portable extensions*, Section 6.1, “The Contextual interface” and Section 6.2, “The Context interface” define an SPI that allows portable extensions to integrate with the container.

1.2. Relationship to other specifications

An application developer creates container-managed components such as JavaBeans, EJBs or servlets and then provides additional metadata that declares additional behavior defined by this specification. These components may take advantage of the services defined by this specification, together with the enterprise and presentational aspects defined by other Java EE platform technologies.

In addition, this specification defines an SPI that allows alternative, non-platform technologies to integrate with the container and the Java EE environment, for example, alternative web presentation technologies.

1.2.1. Relationship to the Java EE platform specification

In the Java EE 6 environment, all *component classes supporting injection*, as defined by the Java EE 6 platform specification, may inject beans via the dependency injection service.

The Java EE platform specification defines a facility for injecting *resources* that exist in the *Java EE component environment*. Resources are identified by string-based names. This specification bolsters that functionality, adding the ability to inject an open-ended set of object types, including, but not limited to, component environment resources, based upon typesafe qualifiers.

1.2.2. Relationship to EJB

EJB defines a programming model for application components that access transactional resources in a multi-user environment. EJB allows concerns such as role-based security, transaction demarcation, concurrency and scalability to be specified declaratively using annotations and XML deployment descriptors and enforced by the EJB container at runtime.

EJB components may be stateful, but are not by nature contextual. References to stateful component instances must be explicitly passed between clients and stateful instances must be explicitly destroyed by the application.

This specification enhances the EJB component model with contextual lifecycle management.

Any session bean instance obtained via the dependency injection service is a contextual instance. It is bound to a lifecycle context and is available to other objects that execute in that context. The container automatically creates the instance when it is needed by a client. When the context ends, the container automatically destroys the instance.

Message-driven and entity beans are by nature non-contextual objects and may not be injected into other objects.

The container performs dependency injection on all session and message-driven bean instances, even those which are not contextual instances.

1.2.3. Relationship to managed beans

The Managed Beans specification defines the basic programming model for application components managed by the Java EE container.

As defined by this specification, most Java classes, including all JavaBeans, are managed beans.

This specification defines contextual lifecycle management and dependency injection as generic services applicable to all managed beans.

Any managed bean instance obtained via the dependency injection service is a contextual instance. It is bound to a lifecycle context and is available to other objects that execute in that context. The container automatically creates the instance when it is needed by a client. When the context ends, the container automatically destroys the instance.

The container performs dependency injection on all managed bean instances, even those which are not contextual instances.

1.2.4. Relationship to Dependency Injection for Java

The Dependency Injection for Java specification defines a set of annotations for the declaring injected fields, methods and constructors of a bean. The dependency injection service makes use of these annotations.

1.2.5. Relationship to Java Interceptors

The Java Interceptors specification defines the basic programming model and semantics for interceptors. This specification enhances that model by providing the ability to associate interceptors with beans using typesafe interceptor bindings.

1.2.6. Relationship to JSF

JavaServer Faces is a web-tier presentation framework that provides a component model for graphical user interface components and an event-driven interaction model that binds user interface components to objects accessible via Unified EL.

This specification allows any bean to be assigned a name. Thus, a JSF application may take advantage of the sophisticated context and dependency injection model defined by this specification.

1.2.7. Relationship to Bean Validation

Bean Validation provides a unified way of declaring and defining constraints on an object model, defines a runtime engine to validate objects and provides method validation.

The Bean Validation specification defines beans for Bean Validation managed objects including `Validator` and `ValidatorFactory`. A number of Bean Validation managed instances, including `ConstraintValidators` can take advantage of dependency injection. Bean Validation also provides support for method parameter validation on any bean.

1.3. Introductory examples

The following examples demonstrate the use of lifecycle contexts and dependency injection.

1.3.1. JSF example

The following JSF page defines a login prompt for a web application:

```
<f:view>
    <h:form>
        <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
            <h:outputLabel for="username">Username:</h:outputLabel>
            <h:inputText id="username" value="#{credentials.username}"/>
            <h:outputLabel for="password">Password:</h:outputLabel>
            <h:inputText id="password" value="#{credentials.password}"/>
        </h:panelGrid>
        <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
        <h:commandButton value="Logout" action="#{login.logout}" rendered="#{login.loggedIn}"/>
    </h:form>
</f:view>
```

The Unified EL expressions in this page refer to beans named `credentials` and `login`.

The `Credentials` bean has a lifecycle that is bound to the JSF request:

```
@Model
public class Credentials {
```

```
    private String username;
    private String password;

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }

}
```

The `@Model` annotation defined in Section 2.7.3, “Built-in stereotypes” is a *stereotype* that identifies the `Credentials` bean as a model object in an MVC architecture.

The `Login` bean has a lifecycle that is bound to the HTTP session:

```
@SessionScoped @Model
public class Login implements Serializable {

    @Inject Credentials credentials;
    @Inject @Users EntityManager userDatabase;

    private CriteriaQuery<User> query;
    private Parameter<String> usernameParam;
    private Parameter<String> passwordParam;

    private User user;

    @Inject
    void initQuery(@Users EntityManagerFactory emf) {
        CriteriaBuilder cb = emf.getCriteriaBuilder();
        usernameParam = cb.parameter(String.class);
        passwordParam = cb.parameter(String.class);
        query = cb.createQuery(User.class);
        Root<User> u = query.from(User.class);
        query.select(u);
        query.where( cb.equal(u.get(User_.username), usernameParam),
                     cb.equal(u.get(User_.password), passwordParam) );
    }

    public void login() {

        List<User> results = userDatabase.createQuery(query)
            .setParameter(usernameParam, credentials.getUsername())
            .setParameter(passwordParam, credentials.getPassword())
            .getResultList();

        if ( !results.isEmpty() ) {
            user = results.get(0);
        }

    }

    public void logout() {
        user = null;
    }

    public boolean isLoggedIn() {
        return user!=null;
    }

    @Produces @LoggedIn User getCurrentUser() {
        if (user==null) {
            throw new NotLoggedInException();
        }
        else {
            return user;
        }
    }

}
```

The `@SessionScoped` annotation defined in Section 2.4.1, “Built-in scope types” is a *scope type* that specifies the lifecycle of instances of `Login`. Managed beans with this scope must be serializable.

The `@Inject` annotation defined by the Dependency Injection for Java specification identifies an *injected field* which is initialized by the container when the bean is instantiated, or an *initializer method* which is called by the container after the bean is instantiated, with injected parameters.

The `@Users` annotation is a qualifier type defined by the application:

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Users {}
```

The `@LoggedIn` annotation is another qualifier type defined by the application:

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface LoggedIn {}
```

The `@Produces` annotation defined in Section 3.3.2, “Declaring a producer method” identifies the method `getCurrentUser()` as a *producer method*, which will be called whenever another bean in the system needs the currently logged-in user, for example, whenever the `user` attribute of the `DocumentEditor` class is injected by the container:

```
@Model
public class DocumentEditor {

    @Inject Document document;
    @Inject @LoggedIn User currentUser;
    @Inject @Documents EntityManager docDatabase;

    public void save() {
        document.setCreatedBy(currentUser);
        em.persist(document);
    }

}
```

The `@Documents` annotation is another application-defined qualifier type. The use of distinct qualifier types enables the container to distinguish which JPA persistence unit is required.

When the login form is submitted, JSF assigns the entered username and password to an instance of the `Credentials` bean that is automatically instantiated by the container. Next, JSF calls the `login()` method of an instance of `Login` that is automatically instantiated by the container. This instance continues to exist for and be available to other requests in the same HTTP session, and provides the `User` object representing the current user to any other bean that requires it (for example, `DocumentEditor`). If the producer method is called before the `login()` method initializes the user object, it throws a `NotLoggedInException`.

1.3.2. EJB example

Alternatively, we could write our `Login` bean to take advantage of the functionality defined by EJB:

```
@Stateful @SessionScoped @Model
public class Login {

    @Inject Credentials credentials;
    @Inject @Users EntityManager userDatabase;

    ...

    private User user;

    @Inject
    void initQuery(@Users EntityManagerFactory emf) {
       ...
    }

    @TransactionAttribute(REQUIRES_NEW)
    @RolesAllowed("guest")
    public void login() {
        ...
    }

    public void logout() {
        user = null;
    }

    public boolean isLoggedIn() {
        return user!=null;
    }

```

```
    @RolesAllowed("user")
    @Produces @LoggedIn User getCurrentUser() {
        ...
    }

}
```

The EJB `@Stateful` annotation specifies that this bean is an EJB stateful session bean. The EJB `@TransactionAttribute` and `@RolesAllowed` annotations declare the EJB transaction demarcation and security attributes of the annotated methods.

1.3.3. Java EE component environment example

In the previous examples, we injected container-managed persistence contexts using qualifier types. We need to tell the container what persistence context is being referred to by which qualifier type. We can declare references to persistence contexts and other resources in the Java EE component environment in Java code.

```
public class Databases {

    @Produces @PersistenceContext(unitName="UserData")
    @Users EntityManager userDatabaseEntityManager;

    @Produces @PersistenceUnit(unitName="UserData")
    @Users EntityManagerFactory userDatabaseEntityManagerFactory;

    @Produces @PersistenceContext(unitName="DocumentData")
    @Documents EntityManager docDatabaseEntityManager;

}
```

The JPA `@PersistenceContext` and `@PersistenceUnit` annotations identify the JPA persistence unit.

1.3.4. Event example

Beans may raise events. For example, our `Login` class could raise events when a user logs in or out.

```
@SessionScoped @Model
public class Login implements Serializable {

    @Inject Credentials credentials;
    @Inject @Users EntityManager userDatabase;

    @Inject @LoggedIn Event<User> userLoggedInEvent;
    @Inject @LoggedOut Event<User> userLoggedOutEvent;

    ...

    private User user;

    @Inject
    void initQuery(@Users EntityManagerFactory emf) {
       ...
    }

    public void login() {

        List<User> results = ... ;

        if ( !results.isEmpty() ) {
            user = results.get(0);
            userLoggedInEvent.fire(user);
        }

    }

    public void logout() {
        userLoggedOutEvent.fire(user);
        user = null;
    }

    public boolean isLoggedIn() {
        return user!=null;
    }

    @Produces @LoggedIn User getCurrentUser() {
        ...
    }
```

```
}
```

The method `fire()` of the built-in bean of type `Event` defined in Section 10.3.1, “The Event interface” allows the application to fire events. Events consist of an *event object*—in this case the `User`—and event qualifiers. Event qualifiers—such as `@LoggedIn` and `@LoggedOut`—allow event consumers to specify which events of a certain type they are interested in.

Other beans may observe these events and use them to synchronize their internal state, with no coupling to the bean producing the events:

```
@SessionScoped
public class Permissions implements Serializable {

    @Produces
    private Set<Permission> permissions = new HashSet<Permission>();

    @Inject @Users EntityManager userDatabase;
    Parameter<String> usernameParam;
    CriteriaQuery<Permission> query;

    @Inject
    void initQuery(@Users EntityManagerFactory emf) {
        CriteriaBuilder cb = emf.getCriteriaBuilder();
        usernameParam = cb.parameter(String.class);
        query = cb.createQuery(Permission.class);
        Root<Permission> p = query.from(Permission.class);
        query.select(p);
        query.where( cb.equal(p.get(Permission_.user).get(User_.username),
                     usernameParam) );
    }

    void onLogin(@Observes @LoggedIn User user) {
        permissions = new HashSet<Permission>( userDatabase.createQuery(query)
            .setParameter(usernameParam, user.getUsername())
            .getResultList() );
    }

    void onLogout(@Observes @LoggedOut User user {
        permissions.clear();
    }


}
```

The `@Produces` annotation applied to a field identifies the field as a producer field, as defined in Section 3.4, “Producer fields”, a kind of shortcut version of a producer method. This producer field allows the permissions of the current user to be injected to an injection point of type `Set<Permission>`.

The `@Observes` annotation defined in Section 10.4.2, “Declaring an observer method” identifies the method with the annotated parameter as an *observer method* that is called by the container whenever an event matching the type and qualifiers of the annotated parameter is fired.

1.3.5. Injection point metadata example

It is possible to implement generic beans that introspect the injection point to which they belong. This makes it possible to implement injection for `Loggers`, for example.

```
class Loggers {

    @Produces Logger getLogger(InjectionPoint injectionPoint) {
        return Logger.getLogger( injectionPoint.getMember().getDeclaringClass().getSimpleName() );
    }

}
```

The `InjectionPoint` interface defined in Section 5.5.7, “Injection point metadata”, provides metadata about the injection point to the object being injected into it.

Then this class will have a `Logger` named `"Permissions"` injected:

```
@SessionScoped
public class Permissions implements Serializable {

    @Inject Logger log;

```

```
    ...

}
```

1.3.6. Interceptor example

Interceptors allow common, cross-cutting concerns to be applied to beans via custom annotations. Interceptor types may be individually enabled or disabled at deployment time.

The `AuthorizationInterceptor` class defines a custom authorization check:

```
@Secure @Interceptor
public class AuthorizationInterceptor {

    @Inject @LoggedIn User user;
    @Inject Logger log;

    @AroundInvoke
    public Object authorize(InvocationContext ic) throws Exception {
        try {
            if ( !user.isBanned() ) {
                log.fine("Authorized");
                return ic.proceed();
            }
            else {
                log.fine("Not authorized");
                throw new NotAuthorizedException();
            }
        }
        catch (NotAuthenticatedException nae) {
            log.fine("Not authenticated");
            throw nae;
        }
    }

}
```

The `@Interceptor` annotation, defined in Section 9.2, “Declaring the interceptor bindings of an interceptor”, identifies the `AuthorizationInterceptor` class as an interceptor. The `@Secure` annotation is a custom *interceptor binding type*, as defined in Section 9.1, “Interceptor binding types”.

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

The `@Secure` annotation is used to apply the interceptor to a bean:

```
@Model
public class DocumentEditor {

    @Inject Document document;
    @Inject @LoggedIn User user;
    @Inject @Documents EntityManager em;

    @Secure
    public void save() {
        document.setCreatedBy(currentUser);
        em.persist(document);
    }

}
```

When the `save()` method is invoked, the `authorize()` method of the interceptor will be called. The invocation will proceed to the `DocumentEditor` class only if the authorization check is successful.

1.3.7. Decorator example

Decorators are similar to interceptors, but apply only to beans of a particular Java interface. Like interceptors, decorators may be easily enabled or disabled at deployment time. Unlike interceptors, decorators are aware of the semantics of the intercepted method.

For example, the `DataAccess` interface might be implemented by many beans:

```
public interface DataAccess<T, V> {

    public V getId(T object);
    public T load(V id);
    public void save(T object);
    public void delete(T object);

    public Class<T> getDataType();

}
```

The `DataAccessAuthorizationDecorator` class defines the authorization checks:

```
@Decorator
public abstract class DataAccessAuthorizationDecorator<T, V> implements DataAccess<T, V> {

    @Inject @Delegate DataAccess<T, V> delegate;

    @Inject Logger log;
    @Inject Set<Permission> permissions;

    public void save(T object) {
        authorize(SecureAction.SAVE, object);
        delegate.save(object);
    }

    public void delete(T object) {
        authorize(SecureAction.DELETE, object);
        delegate.delete(object);
    }

    private void authorize(SecureAction action, T object) {
        V id = delegate.getId(object);
        Class<T> type = delegate.getDataType();
        if ( permissions.contains( new Permission(action, type, id) ) ) {
            log.fine("Authorized for " + action);
        }
        else {
            log.fine("Not authorized for " + action);
            throw new NotAuthorizedException(action);
        }
    }

}
```

The `@Decorator` annotation defined in Section 8.1.1, “Declaring a decorator” identifies the `DataAccessAuthorization-Decorator` class as a decorator. The `@Delegate` annotation defined in Section 8.1.2, “Decorator delegate injection points” identifies the *delegate*, which the decorator uses to delegate method calls to the container. The decorator applies to any bean that implements `DataAccess`.

The decorator intercepts invocations just like an interceptor. However, unlike an interceptor, the decorator contains functionality that is specific to the semantics of the method being called.

Decorators may be declared abstract, relieving the developer of the responsibility of implementing all methods of the decorated interface. If a decorator does not implement a method of a decorated interface, the decorator will simply not be called when that method is invoked upon the decorated bean.

Chapter 2. Concepts

A Java EE component is a *bean* if the lifecycle of its instances may be managed by the container according to the lifecycle context model defined in Chapter 6, *Scopes and contexts*. A bean may bear metadata defining its lifecycle and interactions with other components.

Speaking more abstractly, a bean is a source of contextual objects which define application state and/or logic. These objects are called *contextual instances of the bean*. The container creates and destroys these instances and associates them with the appropriate context. Contextual instances of a bean may be injected into other objects (including other bean instances) that execute in the same context, and may be used in EL expressions that are evaluated in the same context.

A bean comprises the following attributes:

- A (nonempty) set of bean types
- A (nonempty) set of qualifiers
- A scope
- Optionally, a bean name
- A set of interceptor bindings
- A bean implementation

Furthermore, a bean may or may not be an alternative.

In most cases, a bean developer provides the bean implementation by writing business logic in Java code. The developer then defines the remaining attributes by explicitly annotating the bean class, or by allowing them to be defaulted by the container, as specified in Chapter 3, *Programming model*. In certain other cases—for example, Java EE component environment resources, defined in Section 3.6, “Resources”—the developer provides only the annotations and the bean implementation is provided by the container.

The bean types and qualifiers of a bean determine where its instances will be injected by the container, as defined in Chapter 5, *Dependency injection, lookup and EL*.

The bean developer may also create interceptors and/or decorators or reuse existing interceptors and/or decorators. The interceptor bindings of a bean determine which interceptors will be applied at runtime. The bean types and qualifiers of a bean determine which decorators will be applied at runtime. Interceptors are defined by Java interceptors specification, and interceptor bindings are specified in Chapter 9, *Interceptor bindings*. Decorators are defined in Chapter 8, *Decorators*.

2.1. Functionality provided by the container to the bean

A bean is provided by the container with the following capabilities:

- transparent creation and destruction and scoping to a particular context, specified in Chapter 6, *Scopes and contexts* and Chapter 7, *Lifecycle of contextual instances*,
- scoped resolution by bean type and qualifier annotation type when injected into a Java-based client, as defined by Section 5.2, “Typesafe resolution”,
- scoped resolution by bean name when used in a Unified EL expression, as defined by Section 5.3, “EL name resolution”,
- lifecycle callbacks and automatic injection of other bean instances, specified in Chapter 3, *Programming model* and Chapter 5, *Dependency injection, lookup and EL*,
- method interception, callback interception, and decoration, as defined in Chapter 9, *Interceptor bindings* and Chapter 8, *Decorators*, and
- event notification, as defined in Chapter 10, *Events*.

2.2. Bean types

A bean type defines a client-visible type of the bean. A bean may have multiple bean types. For example, the following bean has four bean types:

```
public class BookShop
        extends Business
        implements Shop<Book> {
    ...
}
```

The bean types are `BookShop`, `Business`, `Shop<Book>` and `Object`.

Meanwhile, this session bean has only the local interfaces `BookShop` and `Auditable`, along with `Object`, as bean types, since the bean class is not a client-visible type.

```
@Stateful
public class BookShopBean
        extends Business
        implements BookShop, Auditable {
    ...
}
```

The rules for determining the (unrestricted) set of bean types for a bean are defined in Section 3.1.2, “Bean types of a managed bean”, Section 3.2.2, “Bean types of a session bean”, Section 3.3.1, “Bean types of a producer method”, Section 3.4.1, “Bean types of a producer field” and Section 3.6.2, “Bean types of a resource”.

All beans have the bean type `java.lang.Object`.

The bean types of a bean are used by the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”.

2.2.1. Legal bean types

Almost any Java type may be a bean type of a bean:

- A bean type may be an interface, a concrete class or an abstract class, and may be declared final or have final methods.
- A bean type may be a parameterized type with actual type parameters and type variables.
- A bean type may be an array type. Two array types are considered identical only if the element type is identical.
- A bean type may be a primitive type. Primitive types are considered to be identical to their corresponding wrapper types in `java.lang`.
- A bean type may be a raw type.

A type variable is not a legal bean type. A parameterized type that contains a wildcard type parameter is not a legal bean type.

Note that certain additional restrictions are specified in Section 3.15, “Unproxyable bean types” for beans with a normal scope, as defined in Section 6.3, “Normal scopes and pseudo-scopes”.

2.2.2. Restricting the bean types of a bean

The bean types of a bean may be restricted by annotating the bean class or producer method or field with the annotation `@javax.enterprise.inject.Typed`.

```
@Typed(Shop.class)
public class BookShop
        extends Business
        implements Shop<Book> {
    ...
}
```

When a `@Typed` annotation is explicitly specified, only the types whose classes are explicitly listed using the `value` member, together with `java.lang.Object`, are bean types of the bean.

In the example, the bean has a two bean types: `Shop<Book>` and `Object`.

If a bean class or producer method or field specifies a `@Typed` annotation, and the `value` member specifies a class which does not correspond to a type in the unrestricted set of bean types of a bean, the container automatically detects the problem and treats it as a definition error.

2.2.3. Typecasting between bean types

A client of a bean may typecast its contextual reference to a bean to any bean type of the bean which is a Java interface. However, the client may not in general typecast its contextual reference to an arbitrary concrete bean type of the bean. For example, if our managed bean was injected to the following field:

```
@Inject Business biz;
```

Then the following typecast is legal:

```
Shop<Book> bookShop = (Shop<Book>) biz;
```

However, the following typecast is not legal and might result in an exception at runtime:

```
BookShop bookShop = (BookShop) biz;
```

2.3. Qualifiers

For a given bean type, there may be multiple beans which implement the type. For example, an application may have two implementations of the interface `PaymentProcessor`:

```
class SynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

```
class AsynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

A client that needs a `PaymentProcessor` that processes payments synchronously needs some way to distinguish between the two different implementations. One approach would be for the client to explicitly specify the class that implements the `PaymentProcessor` interface. However, this approach creates a hard dependence between client and implementation—exactly what use of the interface was designed to avoid!

A *qualifier type* represents some client-visible semantic associated with a type that is satisfied by some implementations of the type (and not by others). For example, we could introduce qualifier types representing synchronicity and asynchronicity. In Java code, qualifier types are represented by annotations.

```
@Synchronous
class SynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

```
@Asynchronous
class AsynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

Finally, qualifier types are applied to injection points to distinguish which implementation is required by the client. For example, when the container encounters the following injected field, an instance of `SynchronousPaymentProcessor` will be injected:

```
@Inject @Synchronous PaymentProcessor paymentProcessor;
```

But in this case, an instance of `AsynchronousPaymentProcessor` will be injected:

```
@Inject @Asynchronous PaymentProcessor paymentProcessor;
```

The container inspects the qualifier annotations and type of the injected attribute to determine the bean instance to be injected, according to the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”.

An injection point may even specify multiple qualifiers.

Qualifier types are also used as event selectors by event consumers, as defined in Chapter 10, *Events*, and to bind decorators to beans, as specified in Chapter 8, *Decorators*.

2.3.1. Built-in qualifier types

Three standard qualifier types are defined in the package `javax.enterprise.inject`. In addition, the built-in qualifier type `@Named` is defined by the package `javax.inject`.

Every bean has the built-in qualifier `@Any`, even if it does not explicitly declare this qualifier, except for the special `@New` qualified beans defined in Section 3.14, “@New qualified beans”.

If a bean does not explicitly declare a qualifier other than `@Named`, the bean has exactly one additional qualifier, of type `@Default`. This is called the *default qualifier*.

The following declarations are equivalent:

```
@Default
public class Order { ... }
```

```
public class Order { ... }
```

Both declarations result in a bean with two qualifiers: `@Any` and `@Default`.

The following declaration results in a bean with three qualifiers: `@Any`, `@Default` and `@Named("ord")`.

```
@Named("ord")
public class Order { ... }
```

The default qualifier is also assumed for any injection point that does not explicitly declare a qualifier, as defined in Section 3.11, “The default qualifier at injection points”. The following declarations, in which the use of the `@Inject` annotation identifies the constructor parameter as an injection point, are equivalent:

```
public class Order {
    @Inject
    public Order(@Default OrderProcessor processor) { ... }
}
```

```
public class Order {
    @Inject
    public Order(OrderProcessor processor) { ... }
}
```

2.3.2. Defining new qualifier types

A qualifier type is a Java annotation defined as `@Retention(RUNTIME)`. Typically a qualifier type is defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})`.

A qualifier type may be declared by specifying the `@javax.inject.Qualifier` meta-annotation.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Synchronous {}
```

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
```

```
public @interface Asynchronous {}
```

A qualifier type may define annotation members.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
    PaymentMethod value();
}
```

2.3.3. Declaring the qualifiers of a bean

The qualifiers of a bean are declared by annotating the bean class or producer method or field with the qualifier types.

```
@LDAP
class LdapAuthenticator
        implements Authenticator {
    ...
}
```

```
public class Shop {

   @Produces @All
   public List<Product> getAllProducts() { ... }

   @Produces @WishList
   public List<Product> getWishList() { ... }

}
```

Any bean may declare multiple qualifier types.

```
@Synchronous @Reliable
class SynchronousReliablePaymentProcessor
        implements PaymentProcessor {
    ...
}
```

2.3.4. Specifying qualifiers of an injected field

Qualifier types may be applied to injected fields (see Section 3.9, “Injected fields”) to determine the bean that is injected, according to the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”.

```
@Inject @LDAP Authenticator authenticator;
```

A bean may only be injected to an injection point if it has all the qualifiers of the injection point.

```
@Inject @Synchronous @Reliable PaymentProcessor paymentProcessor;
```

```
@Inject @All List<Product> catalog;
```

```
@Inject @WishList List<Product> wishList;
```

2.3.5. Specifying qualifiers of a method or constructor parameter

Qualifier types may be applied to parameters of producer methods, initializer methods, disposer methods, observer methods or bean constructors (see Chapter 3, *Programming model*) to determine the bean instance that is passed when the method is called by the container. The container uses the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution” to determine values for these parameters.

For example, when the container encounters the following producer method, an instance of `SynchronousPaymentProcessor` will be passed to the first parameter and an instance of `AsynchronousPaymentProcessor` will be passed to the second parameter:

```
@Produces
```

```
PaymentProcessor getPaymentProcessor(@Synchronous PaymentProcessor sync,
                                     @Asynchronous PaymentProcessor async) {
    return isSynchronous() ? sync : async;
}
```

2.4. Scopes

Java EE components such as servlets, EJBs and JavaBeans do not have a well-defined *scope*. These components are either:

- *singletons*, such as EJB singleton beans, whose state is shared between all clients,
- *stateless objects*, such as servlets and stateless session beans, which do not contain client-visible state, or
- objects that must be explicitly created and destroyed by their client, such as JavaBeans and stateful session beans, whose state is shared by explicit reference passing between clients.

Scoped objects, by contrast, exist in a well-defined lifecycle context:

- they may be automatically created when needed and then automatically destroyed when the context in which they were created ends, and
- their state is automatically shared by clients that execute in the same context.

All beans have a scope. The scope of a bean determines the lifecycle of its instances, and which instances of the bean are visible to instances of other beans, as defined in Chapter 6, *Scopes and contexts*. A scope type is represented by an annotation type.

For example, an object that represents the current user is represented by a session scoped object:

```
@Produces @SessionScoped User getCurrentUser() { ... }
```

An object that represents an order is represented by a conversation scoped object:

```
@ConversationScoped
public class Order { ... }
```

A list that contains the results of a search screen might be represented by a request scoped object:

```
@Produces @RequestScoped @Named("orders")
List<Order> getOrderSearchResults() { ... }
```

The set of scope types is extensible.

2.4.1. Built-in scope types

There are five standard scope types defined by this specification, all defined in the package `javax.enterprise.context`.

- The `@RequestScoped`, `@ApplicationScoped` and `@SessionScoped` annotations defined in Section 6.7, “Context management for built-in scopes” represent the standard scopes defined by the Java Servlets specification.
- The `@ConversationScoped` annotation represents the conversation scope defined in Section 6.7.4, “Conversation context lifecycle”.
- Finally, there is a `@Dependent` pseudo-scope for dependent objects, as defined in Section 6.4, “Dependent pseudo-scope”.

If an interceptor or decorator has any scope other than `@Dependent`, non-portable behavior results.

2.4.2. Defining new scope types

A scope type is a Java annotation defined as `@Retention(RUNTIME)`. Typically a scope type is defined as `@Target({TYPE, METHOD, FIELD})`. All scope types must also specify the `@javax.inject.Scope` or

`@javax.enterprise.context.NormalScope` meta-annotation.

A scope type must not have any attributes. If a scope type has attributes non-portable behavior results.

For example, the following annotation declares a "business process scope":

```
@Inherited
@NormalScope
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface BusinessProcessScoped {}
```

Custom scopes are normally defined by portable extensions, which must also provide a *context object*, as defined in Section 6.2, “The Context interface”, that implements the custom scope.

2.4.3. Declaring the bean scope

The scope of a bean is defined by annotating the bean class or producer method or field with a scope type.

A bean class or producer method or field may specify at most one scope type annotation. If a bean class or producer method or field specifies multiple scope type annotations, the container automatically detects the problem and treats it as a definition error.

```
public class Shop {

   @Produces @ApplicationScoped @All
   public List<Product> getAllProducts() { ... }

   @Produces @SessionScoped @WishList
   public List<Product> getWishList() { ..... }

}
```

Likewise, a bean with the custom business process scope may be declared by annotating it with the `@BusinessProcessScoped` annotation:

```
@BusinessProcessScoped
public class Order { ... }
```

Alternatively, a scope type may be specified using a stereotype annotation, as defined in Section 2.7.2, “Declaring the stereotypes for a bean”.

2.4.4. Default scope

When no scope is explicitly declared by annotating the bean class or producer method or field the scope of a bean is defaulted.

The *default scope* for a bean which does not explicitly declare a scope depends upon its declared stereotypes:

- If the bean does not declare any stereotype with a declared default scope, the default scope for the bean is `@Dependent`.
- If all stereotypes declared by the bean that have some declared default scope have the same default scope, then that scope is the default scope for the bean.
- If there are two different stereotypes declared by the bean that declare different default scopes, then there is no default scope and the bean must explicitly declare a scope. If it does not explicitly declare a scope, the container automatically detects the problem and treats it as a definition error.

If a bean explicitly declares a scope, any default scopes declared by stereotypes are ignored.

2.5. Bean names

A bean may have a *bean name*. A bean with a name may be referred to by its name in Unified EL expressions. A valid bean name is a period-separated list of valid EL identifiers.

The following strings are valid bean names:

```
org.mydomain.myapp.settings
```

```
orderManager
```

There is no relationship between the bean name of a session bean and the EJB name of the bean.

Subject to the restrictions defined in Section 5.3.1, “Ambiguous EL names”, multiple beans may share the same bean name.

Bean names allow the direct use of beans in JSP or JSF pages, as defined in Section 12.5, “Integration with Unified EL”. For example, a bean with the name `products` could be used like this:

```
<h:outputText value="#{products.total}"/>
```

Bean names are used by the rules of EL name resolution defined in Section 5.3, “EL name resolution”.

2.5.1. Declaring the bean name

To specify the name of a bean, the qualifier `@javax.inject.Named` is applied to the bean class or producer method or field. This bean is named `currentOrder`:

```
@Named("currentOrder")
public class Order { ... }
```

2.5.2. Default bean names

In the following circumstances, a *default name* must be assigned by the container:

- A bean class or producer method or field of a bean declares a `@Named` annotation and no bean name is explicitly specified by the `value` member.
- A bean declares a stereotype that declares an empty `@Named` annotation, and the bean does not explicitly specify a bean name.

If a bean class or producer method or field of a bean declares a `@Named` annotation and no bean name is explicitly specified the value of the `value` member is defaulted.

The default name for a bean depends upon the kind of the bean. The rules for determining the default name for a bean are defined in Section 3.1.5, “Default bean name for a managed bean”, Section 3.2.5, “Default bean name for a session bean”, Section 3.3.4, “Default bean name for a producer method” and Section 3.4.3, “Default bean name for a producer field”.

2.5.3. Beans with no name

If `@Named` is not declared by the bean, nor by its stereotypes, a bean has no name.

If an interceptor or decorator has a name, non-portable behavior results.

2.6. Alternatives

An *alternative* is a bean that must be explicitly declared in the `beans.xml` file if it should be available for lookup, injection or EL resolution.

2.6.1. Declaring an alternative

An alternative may be declared by annotating the bean class or producer method or field with the `@Alternative` annotation.

```
@Alternative
public class MockOrder extends Order { ... }
```

Alternatively, an alternative may be declared by annotating a bean, producer method or producer field with a stereotype that declares an `@Alternative` annotation.

If an interceptor or decorator is an alternative, non-portable behavior results.

2.7. Stereotypes

In many systems, use of architectural patterns produces a set of recurring bean roles. A *stereotype* allows a framework developer to identify such a role and declare some common metadata for beans with that role in a central place.

A stereotype encapsulates any combination of:

- a default scope, and
- a set of interceptor bindings.

A stereotype may also specify that:

- all beans with the stereotype have defaulted bean names, or that
- all beans with the stereotype are alternatives.

A bean may declare zero, one or multiple stereotypes.

2.7.1. Defining new stereotypes

A bean stereotype is a Java annotation defined as `@Retention(RUNTIME)`. Typically a bean stereotype is defined as `@Target({TYPE, METHOD, FIELD})`, `@Target(TYPE)`, `@Target(METHOD)`, `@Target(FIELD)` or `@Target({METHOD, FIELD})`.

A stereotype may be declared by specifying the `@javax.enterprise.inject.Stereotype` meta-annotation.

```
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

2.7.1.1. Declaring the default scope for a stereotype

The default scope of a stereotype is defined by annotating the stereotype with a scope type. A stereotype may declare at most one scope. If a stereotype declares more than one scope, the container automatically detects the problem and treats it as a definition error.

For example, the following stereotype might be used to identify action classes in a web application:

```
@RequestScoped
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

Then actions would have scope `@RequestScoped` unless the scope is explicitly specified by the bean.

2.7.1.2. Specifying interceptor bindings for a stereotype

The interceptor bindings of a stereotype are defined by annotating the stereotype with the interceptor binding types. A stereotype may declare zero, one or multiple interceptor bindings, as defined in Section 9.1.2, “Interceptor bindings for stereotypes”.

We may specify interceptor bindings that apply to all actions:

```
@RequestScoped
@Secure
@Transactional
@Stereotype
```

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

2.7.1.3. Declaring a `@Named` stereotype

A stereotype may declare an empty `@Named` annotation, which specifies that every bean with the stereotype has a defaulted name when a name is not explicitly specified by the bean. A `@Named` qualifier declared by a stereotype is not added to the qualifiers of a bean with the stereotype.

If a stereotype declares a non-empty `@Named` annotation, the container automatically detects the problem and treats it as a definition error.

We may specify that all actions have bean names:

```
@RequestScoped
@Secure
@Transactional
@Named
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

A stereotype should not declare any qualifier annotation other than `@Named`. If a stereotype declares any other qualifier annotation, non-portable behavior results.

A stereotype should not be annotated `@Typed`. If a stereotype is annotated `@Typed`, non-portable behavior results.

2.7.1.4. Declaring an `@Alternative` stereotype

A stereotype may declare an `@Alternative` annotation, which specifies that every bean with the stereotype is an alternative.

We may specify that all mock objects are alternatives:

```
@Alternative
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {}
```

2.7.1.5. Stereotypes with additional stereotypes

A stereotype may declare other stereotypes.

```
@Auditable
@Action
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface AuditableAction {}
```

Stereotype declarations are transitive—a stereotype declared by a second stereotype is inherited by all beans and other stereotypes that declare the second stereotype.

Stereotypes declared `@Target(TYPE)` may not be applied to stereotypes declared `@Target({TYPE, METHOD, FIELD})`, `@Target(METHOD)`, `@Target(FIELD)` or `@Target({METHOD, FIELD})`.

2.7.2. Declaring the stereotypes for a bean

Stereotype annotations may be applied to a bean class or producer method or field.

```
@Action
public class LoginAction { ... }
```

The default scope declared by the stereotype may be overridden by the bean:

```
@Mock @ApplicationScoped @Action
public class MockLoginAction extends LoginAction { ... }
```

Multiple stereotypes may be applied to the same bean:

```
@Dao @Action
public class LoginAction { ... }
```

2.7.3. Built-in stereotypes

The built-in stereotype `@javax.enterprise.inject.Model` is intended for use with beans that define the *model* layer of an MVC web application architecture such as JSF:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```

In addition, the special-purpose `@Interceptor` and `@Decorator` stereotypes are defined in Section 9.2, “Declaring the interceptor bindings of an interceptor” and Section 8.1.1, “Declaring a decorator”.

2.8. Problems detected automatically by the container

When the application violates a rule defined by this specification, the container automatically detects the problem. There are three kinds of problem:

- Definition errors—occur when a single bean definition violates the rules of this specification. If a definition error exists, the container must throw a subclass of `javax.enterprise.inject.spi.DefinitionException`.
- Deployment problems—occur when there are problems resolving dependencies, or inconsistent specialization, in a particular deployment. If a deployment problem occurs, the container must throw a subclass of `javax.enterprise.inject.spi.DeploymentException`.
- Exceptions—occur at runtime

Definition errors are *developer errors*. They may be detected by tooling at development time, and are also detected by the container at initialization time. If a definition error exists in a deployment, initialization will be aborted by the container.

Deployment problems are detected by the container at initialization time. If a deployment problem exists in a deployment, initialization will be aborted by the container.

The container is permitted to define a non-portable mode, for use at development time, in which some definition errors and deployment problems do not cause application initialization to abort.

Exceptions represent problems that may not be detected until they actually occur at runtime. All exceptions defined by this specification are unchecked exceptions. All exceptions defined by this specification may be safely caught and handled by the application.

2.9. Priority ordering

Interceptors and decorators are called in a defined order. The order is specified using a integer priority. Lower priority interceptors and decorators are called first. The order of more than one interceptor or decorator with the same priority is undefined.

It is recommended that a priority of:

0 - 99
: is reserved for interceptors and decorators defined by this specification or the Java EE platform specification

100 - 999

is reserved for interceptors and decorators defined by extension libraries

1000 - 1999
: is reserved for interceptors and decorators defined by applications

2000 - 2999
: is reserved for interceptors and decorators defined by extension libraries

3000 - 3099
: is reserved for interceptors and decorators defined by this specification or the Java EE platform specification

OPEN ISSUE: Is priority the right name for this?

Chapter 3. Programming model

The container provides built-in support for injection and contextual lifecycle management of the following kinds of bean:

- Managed beans
- Session beans
- Producer methods and fields
- Resources (Java EE resources, persistence contexts, persistence units, remote EJBs and web services)

All containers must support managed beans, producer methods and producer fields. Java EE and embeddable EJB containers are required by the Java EE and EJB specifications to support EJB session beans and the Java EE component environment. Other containers are not required to provide support for injection or lifecycle management of session beans or resources.

A portable extension may provide other kinds of beans by implementing the interface `Bean` defined in Section 11.1, “The Bean interface”.

3.1. Managed beans

A *managed bean* is a bean that is implemented by a Java class. This class is called the *bean class* of the managed bean. The basic lifecycle and semantics of managed beans are defined by the Managed Beans specification.

If the bean class of a managed bean is annotated with both the `@Interceptor` and `@Decorator` stereotypes, the container automatically detects the problem and treats it as a definition error.

If a managed bean has a non-static public field, it must have scope `@Dependent`. If a managed bean with a non-static public field declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

If the managed bean class is a generic type, it must have scope `@Dependent`. If a managed bean with a parameterized bean class declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

3.1.1. Which Java classes are managed beans?

A top-level Java class is a managed bean if it is defined to be a managed bean by any other Java EE specification, or if it meets all of the following conditions:

- It is not a non-static inner class.
- It is a concrete class, or is annotated `@Decorator`.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in `ejb-jar.xml`.
- It does not implement `javax.enterprise.inject.spi.Extension`.
- It is not annotated `@Vetoed`.
- It has an appropriate constructor—either:
  - the class has a constructor with no parameters, or
  - the class declares a constructor annotated `@Inject`.

All Java classes that meet these conditions are managed beans and thus no special declaration is required to define a managed bean.

3.1.2. Bean types of a managed bean

The unrestricted set of bean types for a managed bean contains the bean class, every superclass and all interfaces it implements directly or indirectly.

Note the additional restrictions upon bean types of beans with normal scopes defined in Section 3.15, “Unproxyable bean types”.

3.1.3. Declaring a managed bean

A managed bean with a constructor that takes no parameters does not require any special annotations. The following classes are beans:

```
public class Shop { .. }
```

```
class PaymentProcessorImpl implements PaymentProcessor { ... }
```

If the managed bean does not have a constructor that takes no parameters, it must have a constructor annotated `@Inject`. No additional special annotations are required.

A bean class may specify a scope, bean name, stereotypes and/or qualifiers:

```
@ConversationScoped @Default
public class ShoppingCart { ... }
```

A managed bean may extend another managed bean:

```
@Named("loginAction")
public class LoginAction { ... }
```

```
@Mock
@Named("loginAction")
public class MockLoginAction extends LoginAction { ... }
```

The second bean is a "mock object" that overrides the implementation of `LoginAction` when running in an embedded EJB Lite based integration testing environment.

3.1.4. Specializing a managed bean

If a bean class of a managed bean X is annotated `@Specializes`, then the bean class of X must directly extend the bean class of another managed bean Y. Then X *directly specializes* Y, as defined in Section 4.3, “Specialization”.

If the bean class of X does not directly extend the bean class of another managed bean, the container automatically detects the problem and treats it as a definition error.

For example, `MockLoginAction` directly specializes `LoginAction`:

```
public class LoginAction { ... }
```

```
@Mock @Specializes
public class MockLoginAction extends LoginAction { ... }
```

3.1.5. Default bean name for a managed bean

The default name for a managed bean is the unqualified class name of the bean class, after converting the first character to lower case.

For example, if the bean class is named `ProductList`, the default bean name is `productList`.

3.2. Session beans

A *session bean* is a bean that is implemented by a session bean with an EJB 3.x client view that is not annotated with `@Vetoed`. The basic lifecycle and semantics of EJB session beans are defined by the EJB specification.

A stateless session bean must belong to the `@Dependent` pseudo-scope. A singleton bean must belong to either the `@ApplicationScoped` scope or to the `@Dependent` pseudo-scope. If a session bean specifies an illegal scope, the container automatically detects the problem and treats it as a definition error. A stateful session bean may have any scope.

When a contextual instance of a session bean is obtained via the dependency injection service, the behavior of `SessionContext.getInvokedBusinessInterface()` is specific to the container implementation. Portable applications should not rely upon the value returned by this method.

If the bean class of a session bean is annotated `@Interceptor` or `@Decorator`, the container automatically detects the problem and treats it as a definition error.

If the session bean class is a generic type, it must have scope `@Dependent`. If a session bean with a parameterized bean class declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

3.2.1. EJB remove methods of session beans

If a session bean is a stateful session bean:

- If the scope is `@Dependent`, the application *may* call any EJB remove method of a contextual instance of the session bean.
- Otherwise, the application *may not* directly call any EJB remove method of any contextual instance of the session bean.

If the application directly calls an EJB remove method of a contextual instance of a session bean that is a stateful session bean and declares any scope other than `@Dependent`, an `UnsupportedOperationException` is thrown.

If the application directly calls an EJB remove method of a contextual instance of a session bean that is a stateful session bean and has scope `@Dependent` then no parameters are passed to the method by the container. Furthermore, the container ignores the instance instead of destroying it when `Contextual.destroy()` is called, as defined in Section 7.3.2, “Lifecycle of stateful session beans”.

3.2.2. Bean types of a session bean

The unrestricted set of bean types for a session bean contains all local interfaces of the bean and their superinterfaces. If the session bean has a bean class local view, the unrestricted set of bean types contains the bean class and all superclasses. In addition, `java.lang.Object` is a bean type of every session bean.

Remote interfaces are not included in the set of bean types.

3.2.3. Declaring a session bean

A session bean does not require any special annotations apart from the component-defining annotation (or XML declaration) required by the EJB specification. The following EJBs are beans:

```
@Singleton
class Shop { .. }
```

```
@Stateless
class PaymentProcessorImpl implements PaymentProcessor { ... }
```

A bean class may also specify a scope, bean name, stereotypes and/or qualifiers:

```
@ConversationScoped @Stateful @Default @Model
public class ShoppingCart { ... }
```

A session bean class may extend another bean class:

```
@Stateless
@Named("loginAction")
public class LoginActionImpl implements LoginAction { ... }
```

```
@Stateless
@Mock
@Named("loginAction")
public class MockLoginActionImpl extends LoginActionImpl { ... }
```

3.2.4. Specializing a session bean

If a bean class of a session bean X is annotated `@Specializes`, then the bean class of X must directly extend the bean class of another session bean Y. Then X *directly specializes* Y, as defined in Section 4.3, “Specialization”.

If the bean class of X does not directly extend the bean class of another session bean, the container automatically detects the problem and treats it as a definition error.

For example, `MockLoginActionBean` directly specializes `LoginActionBean`:

```
@Stateless
public class LoginActionBean implements LoginAction { ... }
```

```
@Stateless @Mock @Specializes
public class MockLoginActionBean extends LoginActionBean { ... }
```

3.2.5. Default bean name for a session bean

The default name for a session bean is the unqualified class name of the session bean class, after converting the first character to lower case.

For example, if the bean class is named `ProductList`, the default bean name is `productList`.

3.3. Producer methods

A *producer method* acts as a source of objects to be injected, where:

- the objects to be injected are not required to be instances of beans, or
- the concrete type of the objects to be injected may vary at runtime, or
- the objects require some custom initialization that is not performed by the bean constructor.

A producer method must be a default-access, public, protected or private, non-abstract method of a managed bean class or session bean class. A producer method may be either static or non-static. If the bean is a session bean, the producer method must be either a business method of the EJB or a static method of the bean class.

If a producer method sometimes returns a null value, then the producer method must have scope `@Dependent`. If a producer method returns a null value at runtime, and the producer method declares any other scope, an `IllegalProductException` is thrown by the container. This restriction allows the container to use a client proxy, as defined in Section 5.4, “Client proxies”.

If the producer method return type is a parameterized type, it must specify an actual type parameter or type variable for each type parameter.

If a producer method return type contains a wildcard type parameter the container automatically detects the problem and treats it as a definition error.

If the producer method return type is a parameterized type with a type variable, it must have scope `@Dependent`. If a producer method with a parameterized return type with a type variable declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

If a producer method return type is a type variable the container automatically detects the problem and treats it as a definition error.

The application may call producer methods directly. However, if the application calls a producer method directly, no parameters will be passed to the producer method by the container; the returned object is not bound to any context; and its lifecycle is not managed by the container.

A bean may declare multiple producer methods.

3.3.1. Bean types of a producer method

The bean types of a producer method depend upon the method return type:

- If the return type is an interface, the unrestricted set of bean types contains the return type, all interfaces it extends directly or indirectly and `java.lang.Object`.
- If a return type is primitive or is a Java array type, the unrestricted set of bean types contains exactly two types: the method return type and `java.lang.Object`.
- If the return type is a class, the unrestricted set of bean types contains the return type, every superclass and all interfaces it implements directly or indirectly.

Note the additional restrictions upon bean types of beans with normal scopes defined in Section 3.15, “Unproxyable bean types”.

3.3.2. Declaring a producer method

A producer method may be declared by annotating a method with the `@javax.enterprise.inject.Produces` annotation.

```
public class Shop {
   @Produces PaymentProcessor getPaymentProcessor() { ... }
   @Produces List<Product> getProducts() { ... }
}
```

A producer method may also specify scope, bean name, stereotypes and/or qualifiers.

```
public class Shop {
   @Produces @ApplicationScoped @Catalog @Named("catalog")
   List<Product> getProducts() { ... }
}
```

If a producer method is annotated `@Inject`, has a parameter annotated `@Disposes`, or has a parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error.

If a non-static method of a session bean class is annotated `@Produces`, and the method is not a business method of the session bean, the container automatically detects the problem and treats it as a definition error.

Interceptors and decorators may not declare producer methods. If an interceptor or decorator has a method annotated `@Produces`, the container automatically detects the problem and treats it as a definition error.

A producer method may have any number of parameters. All producer method parameters are injection points.

```
public class OrderFactory {

   @Produces @ConversationScoped
   public Order createCurrentOrder(Shop shop, @Selected Product product) {
       Order order = new Order(product, shop);
       return order;
   }

}
```

3.3.3. Specializing a producer method

If a producer method X is annotated `@Specializes`, then it must be non-static and directly override another producer method Y. Then X *directly specializes* Y, as defined in Section 4.3, “Specialization”.

If the method is static or does not directly override another producer method, the container automatically detects the problem and treats it as a definition error.

```
@Mock
```

```
public class MockShop extends Shop {

   @Override @Specializes
   @Produces
   PaymentProcessor getPaymentProcessor() {
      return new MockPaymentProcessor();
   }

   @Override @Specializes
   @Produces
   List<Product> getProducts() {
      return PRODUCTS;
   }

   ...

}
```

3.3.4. Default bean name for a producer method

The default name for a producer method is the method name, unless the method follows the JavaBeans property getter naming convention, in which case the default name is the JavaBeans property name.

For example, this producer method is named `products`:

```
@Produces @Named
public List<Product> getProducts() { ... }
```

This producer method is named `paymentProcessor`:

```
@Produces @Named
public PaymentProcessor paymentProcessor() { ... }
```

3.4. Producer fields

A *producer field* is a slightly simpler alternative to a producer method.

A producer field must be a default-access, public, protected or private, field of a managed bean class or session bean class. A producer field may be either static or non-static. If the bean is a session bean, the producer field must be a static field of the bean class.

If a producer field sometimes contains a null value when accessed, then the producer field must have scope `@Dependent`. If a producer field contains a null value at runtime, and the producer field declares any other scope, an `IllegalProductException` is thrown by the container. This restriction allows the container to use a client proxy, as defined in Section 5.4, “Client proxies”.

If the producer field type is a parameterized type, it must specify an actual type parameter or type variable for each type parameter.

If a producer field type contains a wildcard type parameter the container automatically detects the problem and treats it as a definition error.

If the producer field type is a parameterized type with a type variable, it must have scope `@Dependent`. If a producer field with a parameterized type with a type variable declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

If a producer field type is a type variable the container automatically detects the problem and treats it as a definition error.

The application may access producer fields directly. However, if the application accesses a producer field directly, the returned object is not bound to any context; and its lifecycle is not managed by the container.

A bean may declare multiple producer fields.

3.4.1. Bean types of a producer field

The bean types of a producer field depend upon the field type:

- If the field type is an interface, the unrestricted set of bean types contains the field type, all interfaces it extends directly or indirectly and `java.lang.Object`.
- If a field type is primitive or is a Java array type, the unrestricted set of bean types contains exactly two types: the field type and `java.lang.Object`.
- If the field type is a class, the unrestricted set of bean types contains the field type, every superclass and all interfaces it implements directly or indirectly.

Note the additional restrictions upon bean types of beans with normal scopes defined in Section 3.15, “Unproxyable bean types”.

3.4.2. Declaring a producer field

A producer field may be declared by annotating a field with the `@javax.enterprise.inject.Produces` annotation.

```
public class Shop {
   @Produces PaymentProcessor paymentProcessor = ....;
   @Produces List<Product> products = ....;
}
```

A producer field may also specify scope, bean name, stereotypes and/or qualifiers.

```
public class Shop {
   @Produces @ApplicationScoped @Catalog @Named("catalog")
   List<Product> products = ....;
}
```

If a producer field is annotated `@Inject`, the container automatically detects the problem and treats it as a definition error.

If a non-static field of a session bean class is annotated `@Produces`, the container automatically detects the problem and treats it as a definition error.

Interceptors and decorators may not declare producer fields. If an interceptor or decorator has a field annotated `@Produces`, the container automatically detects the problem and treats it as a definition error.

3.4.3. Default bean name for a producer field

The default name for a producer field is the field name.

For example, this producer field is named `products`:

```
@Produces @Named
public List<Product> products = ...;
```

3.5. Disposer methods

A disposer method allows the application to perform customized cleanup of an object returned by a producer method or producer field.

A disposer method must be a default-access, public, protected or private, non-abstract method of a managed bean class or session bean class. A disposer method may be either static or non-static. If the bean is a session bean, the disposer method must be a business method of the EJB or a static method of the bean class.

A bean may declare multiple disposer methods.

3.5.1. Disposed parameter of a disposer method

Each disposer method must have exactly one *disposed parameter*, of the same type as the corresponding producer method return type or producer field type. When searching for disposer methods for a producer method or producer field the container considers the type and qualifiers of the disposed parameter. If a producer method or producer field declared by the same bean class is assignable to the disposed parameter, according to the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”, the container must call this method when destroying any instance returned by that producer method or producer field.

A disposer method may resolve to multiple producer methods or producer fields declared by the bean class, in which case the container must call it when destroying any instance returned by any of these producer methods or producer fields.

3.5.2. Declaring a disposer method

A disposer method may be declared by annotating a parameter `@javax.enterprise.inject.Disposes`. That parameter is the disposed parameter. Qualifiers may be declared by annotating the disposed parameter:

```
public class UserDatabaseEntityManager {

    @Produces @ConversationScoped @UserDatabase
    public EntityManager create(EntityManagerFactory emf) {
        return emf.createEntityManager();
    }

    public void close(@Disposes @UserDatabase EntityManager em) {
        em.close();
    }

}
```

```
public class Resources {

    @PersistenceContext
    @Produces @UserDatabase
    private EntityManager em;

    public void close(@Disposes @UserDatabase EntityManager em) {
        em.close();
    }

}
```

If a method has more than one parameter annotated `@Disposes`, the container automatically detects the problem and treats it as a definition error.

If a disposer method is annotated `@Produces` or `@Inject` or has a parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error.

If a non-static method of a session bean class has a parameter annotated `@Disposes`, and the method is not a business method of the session bean, the container automatically detects the problem and treats it as a definition error.

Interceptors and decorators may not declare disposer methods. If an interceptor or decorator has a method annotated `@Disposes`, the container automatically detects the problem and treats it as a definition error.

In addition to the disposed parameter, a disposer method may declare additional parameters, which may also specify qualifiers. These additional parameters are injection points.

```
public void close(@Disposes @UserDatabase EntityManager em, Logger log) { ... }
```

3.5.3. Disposer method resolution

A disposer method is bound to a producer method or producer field if:

- the producer method or producer field is declared by the same bean class as the disposer method, and
- the producer method or producer field is assignable to the disposed parameter, according to the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution” (using Section 5.2.4, “Assignability of raw and parameterized types”).

If there are multiple disposer methods for a single producer method or producer field, the container automatically detects the problem and treats it as a definition error.

If there is no producer method or producer field declared by the bean class that is assignable to the disposed parameter of a disposer method, the container automatically detects the problem and treats it as a definition error.

3.6. Resources

A *resource* is a bean that represents a reference to a resource, persistence context, persistence unit, remote EJB or web service in the Java EE component environment.

By declaring a resource, we enable an object from the Java EE component environment to be injected by a specifying only its type and qualifiers at the injection point. For example, if `@CustomerDatabase` is a qualifier:

```
@Inject @CustomerDatabase Datasource customerData;
```

```
@Inject @CustomerDatabase EntityManager customerDatabaseEntityManager;
```

```
@Inject @CustomerDatabase EntityManagerFactory customerDatabaseEntityManagerFactory;
```

```
@Inject PaymentService remotePaymentService;
```

The container is not required to support resources with scope other than `@Dependent`. Portable applications should not define resources with any scope other than `@Dependent`.

A resource may not have a bean name.

3.6.1. Declaring a resource

A resource may be declared by specifying a Java EE component environment injection annotation as part of a producer field declaration. The producer field may be static.

- For a Java EE resource, `@Resource` must be specified.
- For a persistence context, `@PersistenceContext` must be specified.
- For a persistence unit, `@PersistenceUnit` must be specified.
- For a remote EJB, `@EJB` must be specified.
- For a web service, `@WebServiceRef` must be specified.

The injection annotation specifies the metadata needed to obtain the resource, entity manager, entity manager factory, remote EJB instance or web service reference from the component environment.

```
@Produces @WebServiceRef(lookup="java:app/service/PaymentService")
PaymentService paymentService;
```

```
@Produces @EJB(ejbLink="../their.jar#PaymentService")
PaymentService paymentService;
```

```
@Produces @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
@CustomerDatabase Datasource customerDatabase;
```

```
@Produces @PersistenceContext(unitName="CustomerDatabase")
@CustomerDatabase EntityManager customerDatabasePersistenceContext;
```

```
@Produces @PersistenceUnit(unitName="CustomerDatabase")
@CustomerDatabase EntityManagerFactory customerDatabasePersistenceUnit;
```

The bean type and qualifiers of the resource are determined by the producer field declaration.

If the producer field declaration specifies a bean name, the container automatically detects the problem and treats it as a definition error.

If the matching object in the Java EE component environment is not of the same type as the producer field declaration, the container automatically detects the problem and treats it as a definition error.

3.6.2. Bean types of a resource

The unrestricted set of bean types of a resource is determined by the declared type of the producer field, as specified by Section 3.4.1, “Bean types of a producer field”.

3.7. Additional built-in beans

A Java EE or embeddable EJB container must provide the following built-in beans, all of which have qualifier `@Default`:

- a bean with bean type `javax.transaction.UserTransaction`, allowing injection of a reference to the JTA `UserTransaction`, and
- a bean with bean type `javax.security.Principal`, allowing injection of a `Principal` representing the current caller identity.

A servlet container must provide the following built-in beans, all of which have qualifier `@Default`:

- a bean with bean type `javax.servlet.http.HttpServletRequest`, allowing injection of a reference to the `HttpServletRequest`
- a bean with bean type `javax.servlet.http.HttpSession`, allowing injection of a reference to the `HttpSession`,
- a bean with bean type `javax.servlet.ServletContext`, allowing injection of a reference to the `ServletContext`,

These beans are passivation capable dependencies, as defined in Section 6.6.2, “Passivation capable dependencies”.

If a Java EE component class has an injection point of type `UserTransaction` and qualifier `@Default`, and may not validly make use of the JTA `UserTransaction` according to the Java EE platform specification, the container automatically detects the problem and treats it as a definition error.

3.8. Bean constructors

When the container instantiates a bean class, it calls the *bean constructor*. The bean constructor is a default-access, public, protected or private constructor of the bean class.

The application may call bean constructors directly. However, if the application directly instantiates the bean, no parameters are passed to the constructor by the container; the returned object is not bound to any context; no dependencies are injected by the container; and the lifecycle of the new instance is not managed by the container.

3.8.1. Declaring a bean constructor

The bean constructor may be identified by annotating the constructor `@Inject`.

```
@SessionScoped
public class ShoppingCart implements Serializable {

   private User customer;

   @Inject
   public ShoppingCart(User customer) {
       this.customer = customer;
   }

   public ShoppingCart(ShoppingCart original) {
       this.customer = original.customer;
   }

   ShoppingCart() {}

   ...

}
```

```
@ConversationScoped
public class Order {
```

```
    private Product product;
    private User customer;

    @Inject
    public Order(@Selected Product product, User customer) {
        this.product = product;
        this.customer = customer;
    }

    public Order(Order original) {
        this.product = original.product;
        this.customer = original.customer;
    }

    Order() {}

    ...

}
```

If a bean class does not explicitly declare a constructor using `@Inject`, the constructor that accepts no parameters is the bean constructor.

If a bean class has more than one constructor annotated `@Inject`, the container automatically detects the problem and treats it as a definition error.

If a bean constructor has a parameter annotated `@Disposes`, or `@Observes`, the container automatically detects the problem and treats it as a definition error.

A bean constructor may have any number of parameters. All parameters of a bean constructor are injection points.

3.9. Injected fields

An *injected field* is a non-static, non-final field of a bean class, a non-static, non-final field of an enum, or of any Java EE component class supporting injection.

3.9.1. Declaring an injected field

An injected field may be declared by annotating the field `@javax.inject.Inject`.

```
@ConversationScoped
public class Order {

    @Inject @Selected Product product;
    @Inject User customer;

}
```

If an injected field is annotated `@Produces`, the container automatically detects the problem and treats it as a definition error.

3.10. Initializer methods

An *initializer method* is a default-access, public, protected or private, non-abstract, non-static, non-generic method of a bean class, a non-abstract, non-static method of an enum or of any Java EE component class supporting injection. If the bean is a session bean, the initializer method is *not* required to be a business method of the session bean.

A bean class may declare multiple (or zero) initializer methods.

Method interceptors are never called when the container calls an initializer method.

The application may call initializer methods directly, but then no parameters will be passed to the method by the container.

3.10.1. Declaring an initializer method

An initializer method may be declared by annotating the method `@javax.inject.Inject`.

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Inject
   void setProduct(@Selected Product product) {
       this.product = product;
   }

   @Inject
   public void setCustomer(User customer) {
       this.customer = customer;
   }

}
```

If a generic method of a bean is annotated `@Inject`, the container automatically detects the problem and treats it as a definition error.

If an initializer method is annotated `@Produces`, has a parameter annotated `@Disposes`, or has a parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error.

An initializer method may have any number of parameters. All initializer method parameters are injection points.

3.11. The default qualifier at injection points

If an injection point declares no qualifier, the injection point has exactly one qualifier, the default qualifier `@Default`.

The following are equivalent:

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Inject
   public void init(@Selected Product product, User customer) {
       this.product = product;
       this.customer = customer;
   }

}
```

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Inject
   public void init(@Selected Product product, @Default User customer) {
       this.product = product;
       this.customer = customer;
   }

}
```

The following definitions are equivalent:

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

   @Inject Payment(Order order) {
      this(order.getAmount();
   }

}
```

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

```

```
   @Inject Payment(@Default Order order) {
      this(order.getAmount();
   }

}
```

Finally, the following are equivalent:

```
@Inject Order order;
```

```
@Inject @Default Order order;
```

3.12. Vetoing types

Any type or package may be prevented from being considered by CDI by adding the `@Vetoed` annotation on the type or package.

```
@Vetoed
    public class Order {
    ...
 }
```

3.13. The qualifier `@Named` at injection points

The use of `@Named` as an injection point qualifier is not recommended, except in the case of integration with legacy code that uses string-based names to identify beans.

If an injected field declares a `@Named` annotation that does not specify the `value` member, the name of the field is assumed. For example, the following field has the qualifier `@Named("paymentService")`:

```
@Inject @Named PaymentService paymentService;
```

If any other injection point declares a `@Named` annotation that does not specify the `value` member, the container automatically detects the problem and treats it as a definition error.

3.14. `@New` qualified beans

The `@New` qualifier was deprecated in CDI 1.1. CDI applications are encouraged to inject `@Dependent` scoped beans instead.

For each managed bean, and for each session bean, a second bean exists which:

- has the same bean class,
- has the same bean types,
- has the same bean constructor, initializer methods and injected fields, and
- has the same interceptor bindings.

However, this second bean:

- has scope `@Dependent`,
- has exactly one qualifier: `@javax.enterprise.inject.New(X.class)` where `X` is the bean class,
- has no bean name,
- has no stereotypes,
- has no observer methods, producer methods or fields or disposer methods, and

- is not an alternative, and
- is enabled, in the sense of Section 5.1.2, “Enabled and disabled beans”, if and only if some other enabled bean has an injection point with the qualifier `@New(X.class)` where `X` is the bean class.

This bean is called the *`@New` qualified bean* for the class `X`.

Note that this second bean exists—and may be enabled and available for injection—even if the first bean is disabled, as defined by Section 5.1.2, “Enabled and disabled beans”, or if the bean class is deployed outside of a bean archive, as defined in Section 12.1, “Bean archives”, and is therefore not discovered during the bean discovery process defined in Chapter 12, *Packaging and deployment*. The container discovers `@New` qualified beans by inspecting injection points of other enabled beans.

This allows the application to obtain a new instance of a bean which is not bound to the declared scope, but has had dependency injection performed.

```
@Produces @ConversationScoped
@Special Order getSpecialOrder(@New(Order.class) Order order) {
    ...
    return order;
}
```

When the qualifier `@New` is specified at an injection point and no `value` member is explicitly specified, the container defaults the `value` to the declared type of the injection point. So the following injection point has qualifier `@New(Order.class)`:

```
@Produces @ConversationScoped
@Special Order getSpecialOrder(@New Order order) { ... }
```

3.15. Unproxyable bean types

The container uses proxies to provide certain functionality. Certain legal bean types cannot be proxied by the container:

- classes which don't have a non-private constructor with no parameters,
- classes which are declared final,
- classes which have non-static, final methods with public, protected or default visibility,
- primitive types,
- and array types.

A bean type must be proxyable if an injection point resolves to a bean:

- that requires a client proxy, or
- that has an associated decorator, or
- that has a bound interceptor.

Otherwise, the container automatically detects the problem, and treats it as a deployment problem.

Chapter 4. Inheritance and specialization

A bean may inherit type-level metadata and members from its superclasses.

Inheritance of type-level metadata by beans from their superclasses is controlled via use of the Java `@Inherited` meta-annotation. Type-level metadata is never inherited from interfaces implemented by a bean.

Member-level metadata is not inherited. However, injected fields, initializer methods, lifecycle callback methods and non-static observer methods are inherited by beans from their superclasses.

The implementation of a bean may be extended by the implementation of a second bean. This specification recognizes two distinct scenarios in which this situation occurs:

- The second bean *specializes* the first bean in certain deployment scenarios. In these deployments, the second bean completely replaces the first, fulfilling the same role in the system.
- The second bean is simply reusing the Java implementation, and otherwise bears no relation to the first bean. The first bean may not even have been designed for use as a contextual object.

The two cases are quite dissimilar.

By default, Java implementation reuse is assumed. In this case, the two beans have different roles in the system, and may both be available in a particular deployment.

The bean developer may explicitly specify that the second bean specializes the first. Then the second bean inherits, and may not override, the qualifiers and bean name of the first bean. The second bean is able to serve the same role in the system as the first. In a particular deployment, only one of the two beans may fulfill that role.

4.1. Inheritance of type-level metadata

Suppose a class X is extended directly or indirectly by the bean class of a managed bean or session bean Y.

- If X is annotated with a qualifier type, stereotype or interceptor binding type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares an annotation of type Z.

  (This behavior is defined by the Java Language Specification.)
- If X is annotated with a scope type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares a scope type.

  (This behavior is different to what is defined in the Java Language Specification.)

A scope type explicitly declared by X and inherited by Y from X takes precedence over default scopes of stereotypes declared or inherited by Y.

For annotations defined by the application or third-party extensions, it is recommended that:

- scope types should be declared `@Inherited`,
- qualifier types should not be declared `@Inherited`,
- interceptor binding types should be declared `@Inherited`, and
- stereotypes may be declared `@Inherited`, depending upon the semantics of the stereotype.

All scope types, qualifier types, and interceptor binding types defined by this specification adhere to these recommendations.

The stereotypes defined by this specification are not declared `@Inherited`.

However, in special circumstances, these recommendations may be ignored.

Note that the @Named annotation is not declared @Inherited and bean names are not inherited unless specialization is used.

4.2. Inheritance of member-level metadata

Suppose a class X is extended directly or indirectly by the bean class of a managed bean or session bean Y.

- If X declares an injected field x then Y inherits x.

  (This behavior is defined by the Common Annotations for the Java Platform specification.)

- If X declares an initializer, non-static observer, @PostConstruct or @PreDestroy method x() then Y inherits x() if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y overrides the method x().

  (This behavior is defined by the Common Annotations for the Java Platform specification.)

- If X declares a non-static method x() annotated with an interceptor binding type Z then Y inherits the binding if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y overrides the method x().

  (This behavior is defined by the Common Annotations for the Java Platform specification.)

- If X declares a non-static producer or disposer method x() then Y does not inherit this method.

  (This behavior is different to what is defined in the Common Annotations for the Java Platform specification.)

- If X declares a non-static producer field x then Y does not inherit this field.

  (This behavior is different to what is defined in the Common Annotations for the Java Platform specification.)

If X is a generic type, and an injection point or observer method declared by X is inherited by Y, and the declared type of the injection point or event parameter contains type variables declared by X, the type of the injection point or event parameter inherited in Y is the declared type, after substitution of actual type arguments declared by Y or any intermediate class that is a subclass of X and a superclass of Y.

For example, the bean DaoClient has an injection point of type Dao<T>.

```
public class DaoClient<T> {

    @Inject Dao<T> dao;
    ...

}
```

This injection point is inherited by UserDaoClient, but the type of the inherited injection point is Dao<User>.

```
public class UserDaoClient
        extends DaoClient<User> { ... }
```

4.3. Specialization

If two beans both support a certain bean type, and share at least one qualifier, then they are both eligible for injection to any injection point with that declared type and qualifier.

Consider the following beans:

```
@Default @Asynchronous
public class AsynchronousService implements Service {
    ...
}
```

```
@Alternative
public class MockAsynchronousService extends AsynchronousService {
    ...
}
```

Suppose that the MockAsynchronousService alternative is declared in the beans.xml file of some bean archive, as defined in Section 5.1, “Modularity”:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
      <class>org.mycompany.mock.MockAsynchronousService</class>
   </alternatives>
</beans>
```

Then, according to the rules of Section 5.2.2, “Unsatisfied and ambiguous dependencies”, the following ambiguous dependency is resolvable, and so the attribute will receive an instance of `MockAsynchronousService`:

```
@Inject Service service;
```

However, the following attribute will receive an instance of `AsynchronousService`, even though `MockAsynchronousService` is a selected alternative, because `MockAsynchronousService` does not have the qualifier `@Asynchronous`:

```
@Inject @Asynchronous Service service;
```

This is a useful behavior in some circumstances, however, it is not always what is intended by the developer.

The only way one bean can completely override a second bean at all injection points is if it implements all the bean types and declares all the qualifiers of the second bean. However, if the second bean declares a producer method or observer method, then even this is not enough to ensure that the second bean is never called!

To help prevent developer error, the first bean may:

- directly extend the bean class of the second bean, or
- directly override the producer method, in the case that the second bean is a producer method, and then

explicitly declare that it *specializes* the second bean.

```
@Alternative @Specializes
public class MockAsynchronousService extends AsynchronousService {
    ...
}
```

When an enabled bean, as defined in Section 5.1.2, “Enabled and disabled beans”, specializes a second bean, we can be certain that the second bean is never instantiated or called by the container. Even if the second bean defines a producer or observer method, the method will never be called.

4.3.1. Direct and indirect specialization

The annotation `@javax.enterprise.inject.Specializes` is used to indicate that one bean *directly specializes* another bean, as defined in Section 3.1.4, “Specializing a managed bean”, Section 3.2.4, “Specializing a session bean” and Section 3.3.3, “Specializing a producer method”.

Formally, a bean X is said to *specialize* another bean Y if either:

- X directly specializes Y, or
- a bean Z exists, such that X directly specializes Z and Z specializes Y.

Then X will inherit the qualifiers and bean name of Y:

- the qualifiers of X include all qualifiers of Y, together with all qualifiers declared explicitly by X, and
- if Y has a bean name, the bean name of X is the same as the bean name of Y.

Furthermore, X must have all the bean types of Y. If X does not have some bean type of Y, the container automatically detects the problem and treats it as a definition error.

If Y has a bean name and X declares a bean name explicitly the container automatically detects the problem and treats it as a definition error.

For example, the following bean would have the inherited qualifiers `@Default` and `@Asynchronous`:

```
@Mock @Specializes
public class MockAsynchronousService extends AsynchronousService {
    ...
}
```

If `AsynchronousService` declared a bean name:

```
@Default @Asynchronous @Named("asyncService")
public class AsynchronousService implements Service{
    ...
}
```

Then the bean name would also automatically be inherited by `MockAsynchronousService`.

If an interceptor or decorator is annotated `@Specializes`, non-portable behavior results.

Chapter 5. Dependency injection, lookup and EL

The container injects references to contextual instances to the following kinds of *injection point*:

- Any injected field of a bean class or enum
- Any parameter of a bean constructor, bean initializer method, enum initializer method, producer method or disposer method
- Any parameter of an observer method, except for the event parameter

References to contextual instances may also be obtained by programmatic lookup or by Unified EL expression evaluation.

In general, a bean type or bean name does not uniquely identify a bean. When resolving a bean at an injection point, the container considers bean type, qualifiers and alternative declarations in `beans.xml`. When resolving a name in an EL expression, the container considers the bean name and alternative declarations in `beans.xml`. This allows bean developers to decouple type from implementation.

The container is required to support circularities in the bean dependency graph where at least one bean participating in every circular chain of dependencies has a normal scope, as defined in Section 6.3, “Normal scopes and pseudo-scopes”. The container is not required to support circular chains of dependencies where every bean participating in the chain has a pseudo-scope.

5.1. Modularity

Beans and their clients may be deployed in *modules* in a module architecture such as the Java EE environment. In a module architecture, certain modules are considered *bean archives*. In the Java EE module architecture, any Java EE module or library is a module. The Java EE module or library is a bean archive if it contains a `beans.xml` file, as defined in Section 12.1, “Bean archives”.

A bean packaged in a certain module is available for injection, lookup and EL resolution to classes and JSP/JSF pages packaged in some other module if and only if the bean class of the bean is required to be *accessible* to the other module by the class accessibility requirements of the module architecture. In the Java EE module architecture, a bean class is accessible in a module if and only if it is required to be accessible according to the class loading requirements defined by the Java EE platform specification.

Note that, in some Java EE implementations, a bean class might be accessible to some other class even when this is not required by the Java EE platform specification. For the purposes of this specification, a class is not considered accessible to another class unless accessibility is explicitly required by the Java EE platform specification.

An alternative is not available for injection, lookup or EL resolution to classes or JSP/JSF pages in a module unless the module is a bean archive and the alternative is explicitly *selected* in that bean archive. An alternative is never available for injection, lookup or EL resolution in a module that is not a bean archive.

5.1.1. Declaring selected alternatives for a bean archive

Alternatives may be selected for an application, or selected only for a particular bean archive.

An alternative may be selected for an application using the `<alternatives>` element of the `beans.xml` file of the bean archive that declares the alternative. The `<alternatives>` element contains a list of bean classes and stereotypes, along with a priority attribute. An alternative is selected for the entire application if either:

- the alternative is a managed bean or session bean and the bean class of the bean is listed, along with a priority attribute, or
- the alternative is a producer method, field or resource, and the bean class that declares the method or field is listed, along with a priority attribute, or
- any `@Alternative` stereotype of the alternative is listed, along with a priority attribute.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
      <class priority="1000">org.mycompany.myfwk.InMemoryDatabase</class>
      <stereotype priority="100">org.mycompany.myfwk.Mock</stereotype>
   </alternatives>
</beans>
```

An alternative may be selected for a bean archive using the `<alternatives>` element of the `beans.xml` file of the bean archive. The `<alternatives>` element contains a list of bean classes and stereotypes, along with a priority attribute. An alternative is selected for the bean archive if either:

- the alternative is a managed bean or session bean and the bean class of the bean is listed, or
- the alternative is a producer method, field or resource, and the bean class that declares the method or field is listed, or
- any `@Alternative` stereotype of the alternative is listed.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
      <stereotype>org.mycompany.site.Australian</stereotype>
   </alternatives>
</beans>
```

An alternative selected for an application may be deselected for a bean archive using the `<alternatives>` element of the `beans.xml` file of the bean archive. The `<alternatives>` element contains a list of bean classes and stereotypes, along with a disabled flag. An alternative is not selected for the bean archive if either:

- the alternative is a managed bean or session bean and the bean class of the bean is listed, along with a disabled flag, or
- the alternative is a producer method, field or resource, and the bean class that declares the method or field is listed, along with a disabled flag, or
- any `@Alternative` stereotype of the alternative is listed, along with a disabled flag.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
      <stereotype enabled="false">org.mycompany.myfwk.Mock</stereotype>
   </alternatives>
</beans>
```

An alternative may be given a default priority, but not selected for an application using the `<alternatives>` element of the `beans.xml` file of the bean archive. The `<alternatives>` element contains a list of bean classes and stereotypes, along with a disabled flag and a priority attribute. An alternative is selected for the bean archive if either:

- the alternative is a managed bean or session bean and the bean class of the bean is listed, along with a disabled flag and a priority attribute, or
- the alternative is a producer method, field or resource, and the bean class that declares the method or field is listed, along with a disabled flag and a priority attribute, or
- any `@Alternative` stereotype of the alternative is listed, along with a disabled flag and a priority attribute.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
      <class priority="1000" enabled="false">org.mycompany.myfwk.InMemoryDatabase</class>
   </alternatives>
</beans>
```

An alternative with a default priority may be selected for a bean archive using the `<alternatives>` element of the `beans.xml` file of the bean archive. The `<alternatives>` element contains a list of bean classes and stereotypes, along with an enabled flag. An alternative is selected for the bean archive if either:

- the alternative is a managed bean or session bean and the bean class of the bean is listed, along with an enabled flag, or
- the alternative is a producer method, field or resource, and the bean class that declares the method or field is listed, along with an enabled flag, or
- any `@Alternative` stereotype of the alternative is listed, along with an enabled flag.

OPEN ISSUE: Can other libraries deselect alternatives and change the priority? If so, which one wins?

OPEN ISSUE: If a library defines a selected or default priority alternative, can it be disabled or enabled for the entire application?

OPEN ISSUE: Should we add a way to specify the priority by annotation?

Each child `<class>` element must specify the name of a bean class of an alternative bean. If there is no bean whose bean class has the specified name, or if no bean whose bean class has the specified name is an alternative, the container automatically detects the problem and treats it as a deployment problem.

Each child `<stereotype>` element must specify the name of an `@Alternative` stereotype annotation. If there is no annotation with the specified name, or the annotation is not an `@Alternative` stereotype, the container automatically detects the problem and treats it as a deployment problem.

If the same type is listed twice under the `<alternatives>` element, the container automatically detects the problem and treats it as a deployment problem.

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `isAlternative()` to determine whether the bean is an alternative, and `getBeanClass()` and `getStereotypes()` to determine whether an alternative is selected in a certain bean archive.

5.1.2. Enabled and disabled beans

A bean is said to be *enabled* if:

- it is deployed in a bean archive, and
- it is not a producer method or field of a disabled bean, and
- it is not specialized by any other enabled bean, as defined in Section 4.3, “Specialization”, and either
- it is not an alternative, or it is a selected alternative of at least one bean archive.

Otherwise, the bean is said to be disabled.

Note that Section 3.14, “@New qualified beans” defines a special rule that determines whether a `@New` qualified bean is enabled or disabled. This rule applies as only to `@New` qualified beans, as an exception to the normal rule defined here.

5.1.3. Inconsistent specialization

Suppose an enabled bean X specializes a second bean Y. If there is another enabled bean that specializes Y we say that *inconsistent specialization* exists. The container automatically detects inconsistent specialization and treats it as a deployment problem.

5.1.4. Inter-module injection

A bean is *available for injection* in a certain module if:

- the bean is not an interceptor or decorator,
- the bean is enabled,
- the bean is either not an alternative, or the module is a bean archive and the bean is a selected alternative of the bean archive, and

- the bean class is required to be accessible to classes in the module, according to the class accessibility requirements of the module architecture.

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `getBeanClass()` to determine the bean class of the bean and `InjectionPoint.getMember()` and then `Member.getDeclaringClass()` to determine the class that declares an injection point.

5.2. Typesafe resolution

The process of matching a bean to an injection point is called *typesafe resolution*. Typesafe resolution usually occurs at application initialization time, allowing the container to warn the user if any enabled beans have unsatisfied or unresolvable ambiguous dependencies.

5.2.1. Performing typesafe resolution

The container considers bean type and qualifiers when resolving a bean to be injected to an injection point. The type and qualifiers of the injection point are called the *required type* and *required qualifiers*.

A bean is *assignable* to a given injection point if:

- The bean has a bean type that matches the required type. For this purpose, primitive types are considered to match their corresponding wrapper types in `java.lang` and array types are considered to match only if their element types are identical. Parameterized and raw types are considered to match if they are identical or if the bean type is *assignable* to the required type, as defined in Section 5.2.4, “Assignability of raw and parameterized types” or Section 8.3.1, “Assignability of raw and parameterized types for delegate injection points”.
- The bean has all the required qualifiers. If no required qualifiers were explicitly specified, the container assumes the required qualifier `@Default`. A bean has a required qualifier if it has a qualifier with (a) the same type and (b) the same annotation member value for each member which is not annotated `@javax.enterprise.util.Nonbinding`.

A bean is eligible for injection to a certain injection point if:

- it is available for injection in the module that contains the class that declares the injection point, and
- it is assignable to the injection point (using Section 5.2.4, “Assignability of raw and parameterized types”).

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `getTypes()` and `getQualifiers()` to determine the bean types and qualifiers.

5.2.2. Unsatisfied and ambiguous dependencies

An *unsatisfied dependency* exists at an injection point when no bean is eligible for injection to the injection point. An *ambiguous dependency* exists at an injection point when multiple beans are eligible for injection to the injection point.

Note that an unsatisfied or ambiguous dependency cannot exist for a decorator delegate injection point, defined in Section 8.1.2, “Decorator delegate injection points”.

When an ambiguous dependency exists, the container attempts to resolve the ambiguity. The container eliminates all eligible beans that are not alternatives, except for producer methods and fields of beans that are alternatives. If there is exactly one bean remaining, the container will select this bean, and the ambiguous dependency is called *resolvable*. If all the beans left are alternatives with a priority, then the container will select the alternative with the highest priority, and the ambiguous dependency is called *resolvable*.

The container must validate all injection points of all enabled beans, all enums, all observer methods, all disposer methods and all other Java EE component classes supporting injection when the application is initialized to ensure that there are no unsatisfied or unresolvable ambiguous dependencies. If an unsatisfied or unresolvable ambiguous dependency exists, the container automatically detects the problem and treats it as a deployment problem.

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `getInjectionPoints()` to determine the set of injection points.

5.2.3. Legal injection point types

Any legal bean type, as defined in Section 2.2.1, “Legal bean types” may be the required type of an injection point. Furthermore, the required type of an injection point may contain a wildcard type parameter. However, a type variable is not a legal injection point type.

If an injection point type is a type variable, the container automatically detects the problem and treats it as a definition error.

5.2.4. Assignability of raw and parameterized types

A parameterized bean type is considered assignable to a raw required type if the raw types are identical and all type parameters of the bean type are either unbounded type variables or `java.lang.Object`.

A parameterized bean type is considered assignable to a parameterized required type if they have identical raw type and for each parameter:

- the required type parameter and the bean type parameter are actual types with identical raw type, and, if the type is parameterized, the bean type parameter is assignable to the required type parameter according to these rules, or
- the required type parameter is a wildcard, the bean type parameter is an actual type and the actual type is assignable to the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the required type parameter is a wildcard, the bean type parameter is a type variable and the upper bound of the type variable is assignable to or assignable from the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the required type parameter is an actual type, the bean type parameter is a type variable and the actual type is assignable from the upper bound, if any, of the type variable, or
- the required type parameter and the bean type parameter are both type variables and the upper bound of the required type parameter is assignable to the upper bound, if any, of the bean type parameter.

For example, `Dao` is eligible for injection to any injection point of type `@Default Dao<Order>`, `@Default Dao<User>`, `@Default Dao<?>`, `@Default Dao<? extends Persistent>` or `@Default Dao<X extends Persistent>` where `X` is a type variable.

```
public class Dao<T extends Persistent> { ... }
```

Furthermore, `UserDao` is eligible for injection to any injection point of type `@Default Dao<User>`, `@Default Dao<?>`, `@Default Dao<? extends Persistent>` or `@Default Dao<? extends User>`.

```
public class UserDao extends Dao<User> { ... }
```

Note that a special set of rules, defined in Section 8.3.1, “Assignability of raw and parameterized types for delegate injection points”, apply if and only if the injection point is a decorator delegate injection point.

5.2.5. Primitive types and null values

For the purposes of typesafe resolution and dependency injection, primitive types and their corresponding wrapper types in the package `java.lang` are considered identical and assignable. If necessary, the container performs boxing or unboxing when it injects a value to a field or parameter of primitive or wrapper type.

However, if an injection point of primitive type resolves to a producer method or producer field that returns a null value at runtime, the container must inject the primitive type's default value as defined by the Java Language Specification.

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `isNullable()` to determine whether the bean may have null values.

5.2.6. Qualifier annotations with members

Qualifier types may have annotation members.

```
@PayBy(CHEQUE) class ChequePaymentProcessor implements PaymentProcessor { ... }
```

```
@PayBy(CREDIT_CARD) class CreditCardPaymentProcessor implements PaymentProcessor { ... }
```

Then only `ChequePaymentProcessor` is a candidate for injection to the following attribute:

```
@Inject @PayBy(CHEQUE) PaymentProcessor paymentProcessor;
```

On the other hand, only `CreditCardPaymentProcessor` is a candidate for injection to this attribute:

```
@Inject @PayBy(CREDIT_CARD) PaymentProcessor paymentProcessor;
```

The container calls the `equals()` method of the annotation member value to compare values.

An annotation member may be excluded from consideration using the `@Nonbinding` annotation.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
    PaymentMethod value();
    @Nonbinding String comment() default "";
}
```

Array-valued or annotation-valued members of a qualifier type should be annotated `@Nonbinding` in a portable application. If an array-valued or annotation-valued member of a qualifier type is not annotated `@Nonbinding`, non-portable behavior results.

5.2.7. Multiple qualifiers

A bean class or producer method or field may declare multiple qualifiers.

```
@Synchronous @PayBy(CHEQUE) class ChequePaymentProcessor implements PaymentProcessor { ... }
```

Then `ChequePaymentProcessor` would be considered a candidate for injection into any of the following attributes:

```
@Inject @PayBy(CHEQUE) PaymentProcessor paymentProcessor;
```

```
@Inject @Synchronous PaymentProcessor paymentProcessor;
```

```
@Inject @Synchronous @PayBy(CHEQUE) PaymentProcessor paymentProcessor;
```

A bean must declare *all* of the qualifiers that are specified at the injection point to be considered a candidate for injection.

5.3. EL name resolution

The process of matching a bean to a name used in EL is called *name resolution*. Since there is no typing information available in EL, the container may consider only the bean name. Name resolution usually occurs at runtime, during EL expression evaluation.

An EL name resolves to a bean if:

- the bean has the given bean name, and
- the bean is available for injection in the war containing the JSP or JSF page with the EL expression.

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `getName()` to determine the bean name.

5.3.1. Ambiguous EL names

An *ambiguous EL name* exists in an EL expression when an EL name resolves to multiple beans. When an ambiguous EL name exists, the container attempts to resolve the ambiguity. The container eliminates all beans that are not alternatives, except for producer methods and fields of beans that are alternatives. If there is exactly one bean remaining, the container will select this bean, and the ambiguous EL name is called *resolvable*. If all the beans left are alternatives with a priority, then the container will select the alternative with the highest priority, and the ambiguous dependency is called *resolvable*.

All unresolvable ambiguous EL names are detected by the container when the application is initialized. Suppose two beans are both available for injection in a certain war, and either:

- the two beans have the same bean name and the name is not resolvable, or
- the bean name of one bean is of the form `x.y`, where `y` is a valid bean name, and `x` is the bean name of the other bean,

the container automatically detects the problem and treats it as a deployment problem.

5.4. Client proxies

An injected reference, or reference obtained by programmatic lookup, is usually a *contextual reference* as defined by Section 6.5.3, “Contextual reference for a bean”.

A contextual reference to a bean with a normal scope, as defined in Section 6.3, “Normal scopes and pseudo-scopes”, is not a direct reference to a contextual instance of the bean (the object returned by `Contextual.create()`). Instead, the contextual reference is a *client proxy* object. A client proxy implements/extends some or all of the bean types of the bean and delegates all method calls to the current instance (as defined in Section 6.3, “Normal scopes and pseudo-scopes”) of the bean.

There are a number of reasons for this indirection:

- The container must guarantee that when any valid injected reference to a bean of normal scope is invoked, the invocation is always processed by the current instance of the injected bean. In certain scenarios, for example if a request scoped bean is injected into a session scoped bean, or into a servlet, this rule requires an indirect reference. (Note that the `@Dependent` pseudo-scope is not a normal scope.)
- The container may use a client proxy when creating beans with circular dependencies. This is only necessary when the circular dependencies are initialized via a managed bean constructor or producer method parameter. (Beans with scope `@Dependent` never have circular dependencies.)
- Finally, client proxies may be passivated, even when the bean itself may not be. Therefore the container must use a client proxy whenever a bean with normal scope is injected into a bean with a passivating scope, as defined in Section 6.6, “Passivation and passivating scopes”. (On the other hand, beans with scope `@Dependent` must be serialized along with their client.)

Client proxies are never required for a bean whose scope is a pseudo-scope such as `@Dependent`.

Client proxies may be shared between multiple injection points. For example, a particular container might instantiate exactly one client proxy object per bean. (However, this strategy is not required by this specification.)

5.4.1. Client proxy invocation

Every time a method of the bean is invoked upon a client proxy, the client proxy must:

- obtain a contextual instance of the bean, as defined in Section 6.5.2, “Contextual instance of a bean”, and
- invoke the method upon this instance.

If the scope is not active, as specified in Section 6.5.1, “The active context object for a scope”, the client proxy rethrows the `ContextNotActiveException` or `IllegalStateException`.

The behavior of all methods declared by `java.lang.Object`, except for `toString()`, is undefined for a client proxy. Portable applications should not invoke any method declared by `java.lang.Object`, except for `toString()`, on a client proxy.

5.5. Dependency injection

From time to time the container instantiates beans and other Java EE component classes supporting injection. The resulting instance may or may not be a *contextual instance* as defined by Section 6.5.2, “Contextual instance of a bean”.

The container is required to perform dependency injection whenever it creates one of the following contextual objects:

- contextual instances of session beans, and
- contextual instances of managed beans.

The container is also required to perform dependency injection whenever it instantiates any of the following non-contextual objects:

- non-contextual instances of session beans (for example, session beans obtained by the application from JNDI or injected using `@EJB`),
- non-contextual instances of managed beans, and
- instances of any other Java EE component class supporting injection.

The container is also required to perform dependency injection at application initialization for enums.

A Java EE 5 container is not required to support injection for non-contextual objects.

The container interacts with instances of beans, enums and other Java EE component classes supporting injection by calling methods and getting and setting field values.

The object injected by the container may not be a direct reference to a contextual instance of the bean. Instead, it is an injectable reference, as defined by Section 6.5.5, “Injectable references”.

5.5.1. Injection using the bean constructor

When the container instantiates a managed bean or session bean with a constructor annotated `@Inject`, the container calls this constructor, passing an injectable reference to each parameter. If there is no constructor annotated `@Inject`, the container calls the constructor with no parameters.

5.5.2. Injection of fields and initializer methods

When the container creates a new instance of a managed bean, session bean, or of any other Java EE component class supporting injection, or injects an enum at application initialization the container must:

- Initialize the values of all injected fields. The container sets the value of each injected field to an injectable reference.
- Call all initializer methods, passing an injectable reference to each parameter.

The container must ensure that:

- Initializer methods declared by a class X in the type hierarchy of the bean are called after all injected fields declared by X or by superclasses of X have been initialized, and after all Java EE component environment resource dependencies declared by X or by superclasses of X have been injected.
- Any `@PostConstruct` callback declared by a class X in the type hierarchy of the bean is called after all initializer methods declared by X or by superclasses of X have been called, after all injected fields declared by X or by superclasses of X have been initialized, and after all Java EE component environment resource dependencies declared by X or by superclasses of X have been injected.
- Any servlet `init()` method is called after all initializer methods have been called, all injected fields have been initialized and all Java EE component environment resource dependencies have been injected.

5.5.3. Destruction of dependent objects

When the container destroys an instance of a bean, enum or of any Java EE component class supporting injection, the container destroys all dependent objects, as defined in Section 6.4.2, "Destruction of objects with scope @Dependent", after the `@PreDestroy` callback completes and after the servlet `destroy()` method is called.

5.5.4. Invocation of producer or disposer methods

When the container calls a producer or disposer method, the behavior depends upon whether the method is static or non-static:

- If the method is static, the container must invoke the method.
- Otherwise, if the method is non-static, the container must:
  - Obtain a contextual instance of the bean which declares the method, as defined by Section 6.5.2, "Contextual instance of a bean".
  - Invoke the method upon this instance, as a business method invocation, as defined in Section 7.2, "Container invocations and interception".

The container passes an injectable reference to each injected method parameter. The container is also responsible for destroying dependent objects created during this invocation, as defined in Section 6.4.2, "Destruction of objects with scope @Dependent".

5.5.5. Access to producer field values

When the container accesses the value of a producer field, the value depends upon whether the field is static or non-static:

- If the producer field is static, the container must access the field value.
- Otherwise, if the producer field is non-static, the container must:
  - Obtain an contextual instance of the bean which declares the producer field, as defined by Section 6.5.2, "Contextual instance of a bean".
  - Access the field value of this instance.

5.5.6. Invocation of observer methods

When the container calls an observer method (defined in Section 10.4, "Observer methods"), the behavior depends upon whether the method is static or non-static:

- If the observer method is static, the container must invoke the method.
- Otherwise, if the observer method is non-static, the container must:
  - Obtain a contextual instance of the bean which declares the observer method according to Section 6.5.2, "Contextual instance of a bean". If this observer method is a conditional observer method, obtain the contextual instance that already exists, only if the scope of the bean that declares the observer method is currently active, without creating a new contextual instance.
  - Invoke the observer method on the resulting instance, if any, as a business method invocation, as defined in Section 7.2, "Container invocations and interception".

The container must pass the event object to the event parameter and an injectable instance to each injected method parameter. The container is also responsible for destroying dependent objects created during this invocation, as defined in Section 6.4.2, "Destruction of objects with scope @Dependent".

5.5.7. Injection point metadata

The interface `javax.enterprise.inject.spi.InjectionPoint` provides access to metadata about an injection point. An instance of `InjectionPoint` may represent:

- an injected field or a parameter of a bean constructor, initializer method, producer method, disposer method or observer method, or
- an instance obtained dynamically using `Instance.get()`.

```
public interface InjectionPoint {
    public Type getType();
    public Set<Annotation> getQualifiers();
    public Bean<?> getBean();
    public Member getMember();
    public Annotated getAnnotated();
    public boolean isDelegate();
    public boolean isTransient();
}
```

- The `getBean()` method returns the `Bean` object representing the bean that defines the injection point. If the injection point does not belong to a bean, `getBean()` returns a null value. If the injection point represents a dynamically obtained instance, the `getBean()` method should return the `Bean` object representing the bean that defines the `Instance` injection point.
- The `getType()` and `getQualifiers()` methods return the required type and required qualifiers of the injection point. If the injection point represents a dynamically obtained instance, the `getType()` and `getQualifiers()` methods should return the required type (as defined by `Instance.select()`), and required qualifiers of the injection point including any additional required qualifiers (as defined by `Instance.select()`).
- The `getMember()` method returns the `Field` object in the case of field injection, the `Method` object in the case of method parameter injection, or the `Constructor` object in the case of constructor parameter injection. If the injection point represents a dynamically obtained instance, the `getMember()` method returns the `Field` object representing the field that defines the `Instance` injection point in the case of field injection, the `Method` object representing the method that defines the `Instance` injection point in the case of method parameter injection, or the `Constructor` object representing the constructor that defines the `Instance` injection point in the case of constructor parameter injection.
- The `getAnnotated()` method returns an instance of `javax.enterprise.inject.spi.AnnotatedField` or `javax.enterprise.inject.spi.AnnotatedParameter`, depending upon whether the injection point is an injected field or a constructor/method parameter. If the injection point represents a dynamically obtained instance, then the `getAnnotated()`method returns an instance of `javax.enterprise.inject.spi.AnnotatedField` or `javax.enterprise.inject.spi.AnnotatedParameter` representing the `Instance` injection point, depending upon whether the injection point is an injected field or a constructor/method parameter.
- The `isDelegate()` method returns `true` if the injection point is a decorator delegate injection point, and `false` otherwise. If the injection point represents a dynamically obtained instance then `isDelegate()` returns false.
- The `isTransient()` method returns `true` if the injection point is a transient field, and `false` otherwise. If the injection point represents a dynamically obtained instance then the `isTransient()` method returns `true` if the `Instance` injection point is a transient field, and `false` otherwise.

Occasionally, a component with scope `@Dependent` needs to access metadata relating to the object into which it is injected. For example, the following producer method creates injectable `Logger`s. The log category of a `Logger` depends upon the class of the object into which it is injected:

```
@Produces Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger( injectionPoint.getMember().getDeclaringClass().getName() );
}
```

The container must provide a bean with scope `@Dependent`, bean type `InjectionPoint` and qualifier `@Default`, allowing dependent objects, as defined in Section 6.4.1, “Dependent objects”, to obtain information about the injection point to which they belong. The built-in implementation must be a passivation capable dependency, as defined in Section 6.6.2, “Passivation capable dependencies”.

If a bean that declares any scope other than `@Dependent` has an injection point of type `InjectionPoint` and qualifier `@Default`, the container automatically detects the problem and treats it as a definition error.

If a Java EE component class supporting injection that is not a bean has an injection point of type `InjectionPoint` and qualifier `@Default`, the container automatically detects the problem and treats it as a definition error.

The `InjectionPoint` injected into a disposer method represents the producer method for which the disposer method is being invoked.

The `InjectionPoint` injected into a decorator represents the injection point on the bean the decorated type is injected into.

5.5.8. Bean metadata

The interfaces `Bean`, `Interceptor` and `Decorator` provide metadata about a bean.

The container must provide beans allowing a bean instance to obtain a `Bean`, `Interceptor` or `Decorator` instance containing its metadata:

- a bean with scope `@Dependent`, qualifier `@Default` and type `Bean` which can be injected into any bean instance
- a bean with scope `@Dependent`, qualifier `@Default` and type `Interceptor` which can be injected into any interceptor instance
- a bean with scope `@Dependent`, qualifier `@Default` and type `Decorator` which can be injected into any decorator instance

Additionally, the container must provide beans allowing interceptors and decorators to obtain information about the beans they intercept and decorate:

- a bean with scope `@Dependent`, qualifier `@Intercepted` and type `Bean` which can be injected into any interceptor instance, and
- a bean with scope `@Dependent`, qualifier `@Decorated` and type `Bean` which can be injected into any decorator instance.

These beans are passivation capable dependencies, as defined in Section 6.6.2, “Passivation capable dependencies”.

If an `Interceptor` instance is injected into a bean instance other than an interceptor instance, the container automatically detects the problem and treats it as a definition error.

If a `Decorator` instance is injected into a bean instance other than a decorator instance, the container automatically detects the problem and treats it as a definition error.

If a `Bean` instance with qualifier `@Intercepted` is injected into a bean instance other than an interceptor instance, the container automatically detects the problem and treats it as a definition error.

If a `Bean` instance with qualifier `@Decorated` is injected into a bean instance other than a decorator instance, the container automatically detects the problem and treats it as a definition error.

The injection of bean metadata is restricted. If:

- the injection point is a field, an initializer method parameter or a bean constructor, with qualifier `@Default`, then the type parameter of the injected `Bean`, `Interceptor` or `Decorator` must be the same as the type declaring the injection point, or
- the injection point is a field, an initializer method parameter or a bean constructor of an interceptor, with qualifier `@Intercepted`, then the type parameter of the injected `Bean` must be an unbounded wildcard, or
- the injection point is a field, an initializer method parameter or a bean constructor of a decorator, with qualifier `@Decorated`, then the type parameter of the injected `Bean` must be the same as the delegate type, or
- the injection point is a producer method parameter then the type parameter of the injected `Bean` must be the same as the producer method return type, or
- the injection point is a disposer method parameter then the type parameter of the injected `Bean` must be the same as the disposed parameter.

Otherwise, the container automatically detects the problem and treats it as a definition error.

If a `Bean` instance is injected into a disposer method, it represents the producer method to which the disposer method is bound.

```
@Named("Order") public class OrderProcessor {

    @Inject Bean<OrderProcessor> bean;

    public void getBeanName() {
       return bean.getName();
    }

}
```

5.6. Programmatic lookup

In certain situations, injection is not the most convenient way to obtain a contextual reference. For example, it may not be used when:

- the bean type or qualifiers vary dynamically at runtime, or
- depending upon the deployment, there may be no bean which satisfies the type and qualifiers, or
- we would like to iterate over all beans of a certain type.

In these situations, an instance of the `javax.enterprise.inject.Instance` interface may be injected:

```
@Inject Instance<PaymentProcessor> paymentProcessor;
```

The method `get()` returns a contextual reference:

```
PaymentProcessor pp = paymentProcessor.get();
```

Any combination of qualifiers may be specified at the injection point:

```
@Inject @PayBy(CHEQUE) Instance<PaymentProcessor> chequePaymentProcessor;
```

Or, the `@Any` qualifier may be used, allowing the application to specify qualifiers dynamically:

```
@Inject @Any Instance<PaymentProcessor> anyPaymentProcessor;
...
Annotation qualifier = synchronously ? new SynchronousQualifier() : new AsynchronousQualifier();
PaymentProcessor pp = anyPaymentProcessor.select(qualifier).get().process(payment);
```

In this example, the returned bean has qualifier `@Synchronous` or `@Asynchronous` depending upon the value of `synchronously`.

Finally, the `@New` qualifier may be used, allowing the application to obtain a `@New` qualified bean, as defined in Section 3.14, “@New qualified beans”:

```
@Inject @New(ChequePaymentProcessor.class) Instance<PaymentProcessor> chequePaymentProcessor;
```

It's even possible to iterate over a set of beans:

```
@Inject @Any Instance<PaymentProcessor> anyPaymentProcessor;
...
for (PaymentProcessor pp: anyPaymentProcessor) pp.test();
```

5.6.1. The `Instance` interface

The `Instance` interface provides a method for obtaining instances of beans with a specified combination of required type and qualifiers, and inherits the ability to iterate beans with that combination of required type and qualifiers from `java.lang.Iterable`:

```
public interface Instance<T> extends Iterable<T>, Provider<T> {

    public Instance<T> select(Annotation... qualifiers);
    public <U extends T> Instance<U> select(Class<U> subtype, Annotation... qualifiers);
    public <U extends T> Instance<U> select(TypeLiteral<U> subtype, Annotation... qualifiers);

    public boolean isUnsatisfied();
    public boolean isAmbiguous();

    public void destroy(T instance);

}
```

For an injected `Instance`:

- the *required type* is the type parameter specified at the injection point, and
- the *required qualifiers* are the qualifiers specified at the injection point.

For example, this injected `Instance` has required type `PaymentProcessor` and required qualifier `@Any`:

```
@Inject @Any Instance<PaymentProcessor> anyPaymentProcessor;
```

The `select()` method returns a child `Instance` for a given required type and additional required qualifiers. If no required type is given, the required type is the same as the parent.

For example, this child `Instance` has required type `AsynchronousPaymentProcessor` and additional required qualifier `@Asynchronous`:

```
Instance<AsynchronousPaymentProcessor> async = anyPaymentProcessor.select(
            AsynchronousPaymentProcessor.class, new AsynchronousQualifier() );
```

If an injection point of raw type `Instance` is defined, the container automatically detects the problem and treats it as a definition error.

If two instances of the same qualifier type are passed to `select()`, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a qualifier type is passed to `select()`, an `IllegalArgumentException` is thrown.

The `get()` method must:

- Identify a bean that has the required type and required qualifiers and is eligible for injection into the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Section 5.2.1, “Performing typesafe resolution”, resolving ambiguities according to Section 5.2.2, “Unsatisfied and ambiguous dependencies”.
- If typesafe resolution results in an unsatisfied dependency, throw an `UnsatisfiedResolutionException`. If typesafe resolution results in an unresolvable ambiguous dependency, throw an `AmbiguousResolutionException`.
- Otherwise, obtain a contextual reference for the bean and the required type, as defined in Section 6.5.3, “Contextual reference for a bean”.

The `iterator()` method must:

- Identify the set of beans that have the required type and required qualifiers and are eligible for injection into the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Section 5.2.1, “Performing typesafe resolution”.
- Return an `Iterator`, that iterates over the set of contextual references for the resulting beans and required type, as defined in Section 6.5.3, “Contextual reference for a bean”.

The method `isUnsatisfied()` returns `true` if there is no bean that has the required type and qualifiers and is eligible for injection into the class into which the parent `Instance` was injected, or `false` otherwise.

The method `isAmbiguous()` returns `true` if there is more than one bean that has the required type and qualifiers and is eligible for injection into the class into which the parent `Instance` was injected, or `false` otherwise.

The method `destroy()` instructs the container to destroy the instance. The bean instance passed to `destroy()` should be a dependent scoped bean instance, or a client proxy for a normal scoped bean. Applications are encouraged to always call `destroy()` when they no longer require an instance obtained from `Instance`.

OPEN ISSUE: If `destroy()` is called on an already destroyed instance, should an `IllegalStateException` occur? Or nothing?

5.6.2. The built-in `Instance`

The container must provide a built-in bean with:

- `Instance<X>` and `Provider<X>` for every legal bean type `X` in its set of bean types,
- every qualifier type in its set of qualifier types,
- scope `@Dependent`,
- no bean name, and
- an implementation provided automatically by the container.

The built-in implementation must be a passivation capable dependency, as defined in Section 6.6.2, “Passivation capable dependencies”.

5.6.3. Using `AnnotationLiteral` and `TypeLiteral`

`javax.enterprise.util.AnnotationLiteral` makes it easier to specify qualifiers when calling `select()`:

```
public PaymentProcessor getSynchronousPaymentProcessor(PaymentMethod paymentMethod) {

    class SynchronousQualifier extends AnnotationLiteral<Synchronous>
            implements Synchronous {}

    class PayByQualifier extends AnnotationLiteral<PayBy>
            implements PayBy {
        public PaymentMethod value() { return paymentMethod; }
    }

    return anyPaymentProcessor.select(new SynchronousQualifier(), new PayByQualifier()).get();
}
```

`javax.enterprise.util.TypeLiteral` makes it easier to specify a parameterized type with actual type parameters when calling `select()`:

```
public PaymentProcessor<Cheque> getChequePaymentProcessor() {
    PaymentProcessor<Cheque> pp = anyPaymentProcessor
        .select( new TypeLiteral<PaymentProcessor<Cheque>>() {} ).get();
}
```

Chapter 6. Scopes and contexts

Associated with every scope type is a *context object*. The context object determines the lifecycle and visibility of instances of all beans with that scope. In particular, the context object defines:

- When a new instance of any bean with that scope is created
- When an existing instance of any bean with that scope is destroyed
- Which injected references refer to any instance of a bean with that scope

The context implementation collaborates with the container via the `Context` and `Contextual` interfaces to create and destroy contextual instances.

6.1. The `Contextual` interface

The interface `javax.enterprise.context.spi.Contextual` defines operations to create and destroy contextual instances of a certain type. Any implementation of `Contextual` is called a *contextual type*. In particular, the `Bean` interface defined in Section 11.1, “The Bean interface” extends `Contextual`, so all beans are contextual types.

```
public interface Contextual<T> {
    public T create(CreationalContext<T> creationalContext);
    public void destroy(T instance, CreationalContext<T> creationalContext);
}
```

- `create()` is responsible for creating new contextual instances of the type.
- `destroy()` is responsible for destroying instances of the type. In particular, it is responsible for destroying all dependent objects of an instance.

If an exception occurs while creating an instance, the exception is rethrown by the `create()` method. If the exception is a checked exception, it must be wrapped and rethrown as an (unchecked) `CreationException`.

If an exception occurs while destroying an instance, the exception must be caught by the `destroy()` method.

If the application invokes a contextual instance after it has been destroyed, the behavior is undefined.

The container and portable extensions may define implementations of the `Contextual` interface that do not extend `Bean`, but it is not recommended that applications directly implement `Contextual`.

6.1.1. The `CreationalContext` interface

The interface `javax.enterprise.context.spi.CreationalContext` provides operations that are used by the `Contextual` implementation during instance creation and destruction.

```
public interface CreationalContext<T> {
    public void push(T incompleteInstance);
    public void release();
}
```

- `push()` registers an *incompletely initialized* contextual instance the with the container. A contextual instance is considered incompletely initialized until it is returned by the `create()` method.
- `release()` destroys all dependent objects, as defined in Section 6.4.1, “Dependent objects”, of the instance which is being destroyed, by passing each dependent object to the `destroy()` method of its `Contextual` object.

The implementation of `Contextual` is not required to call `push()`. However, for certain bean scopes, invocation of `push()` between instantiation and injection helps the container minimize the use of client proxy objects (which would otherwise be required to allow circular dependencies).

If `Contextual.create()` calls `push()`, it must also return the instance passed to `push()`.

`Contextual.create()` should use the given `CreationalContext` when obtaining contextual references to inject, as defined in Section 6.5.3, “Contextual reference for a bean”, in order to ensure that any dependent objects are associated with the contextual instance that is being created.

`Contextual.destroy()` should call `release()` to allow the container to destroy dependent objects of the contextual instance.

6.2. The `Context` interface

The `javax.enterprise.context.spi.Context` interface provides an operation for obtaining contextual instances with a particular scope of any contextual type. Any instance of `Context` is called a context object.

The context object is responsible for creating and destroying contextual instances by calling operations of the `Contextual` interface.

The `Context` interface is called by the container and may be called by portable extensions. It should not be called directly by the application.

```
public interface Context {
    public Class<? extends Annotation> getScope();
    boolean isActive();
    public <T> T get(Contextual<T> bean);
    public <T> T get(Contextual<T> bean, CreationalContext<T> creationalContext);
}
```

```
public interface AlterableContext {
    public void destroy(Contextual<?> contextual);
}
```

The method `getScope()` returns the scope type of the context object.

At a particular point in the execution of the program a context object may be *active* with respect to the current thread. When a context object is active the `isActive()` method returns `true`. Otherwise, we say that the context object is *inactive* and the `isActive()` method returns `false`.

The `get()` method obtains contextual instances of the contextual type represented by the given instance of `Contextual`. The `get()` method may either:

- return an existing instance of the given contextual type, or
- if no `CreationalContext` is given, return a null value, or
- if a `CreationalContext` is given, create a new instance of the given contextual type by calling `Contextual.create()`, passing the given `CreationalContext`, and return the new instance.

If the context object is inactive, the `get()` method must throw a `ContextNotActiveException`.

The `get()` method may not return a null value unless no `CreationalContext` is given, or `Contextual.create()` returns a null value.

The `get()` method may not create a new instance of the given contextual type unless a `CreationalContext` is given.

When the container calls `get()` for a context that is associated with a passivating scope it must ensure that the given instance of `Contextual` and the instance of `CreationalContext`, if given, are serializable.

The `destroy()` method destroys an existing contextual instance, removing it from the context instance.

The `AlterableContext` interface was introduced in Contexts and Dependency Injection for Java EE 1.1 to allow bean instances to be destroyed by the application. Extensions should implement `AlterableContext` instead of `Context`.

The context object is responsible for destroying any contextual instance it creates by passing the instance to the `destroy()` method of the `Contextual` object representing the contextual type. A destroyed instance must not subsequently be returned by the `get()` method.

The context object must pass the same instance of `CreationalContext` to `Contextual.destroy()` that it passed to `Con-`

`textual.create()` when it created the instance.

OPEN ISSUE: What name should the `AlterableContext` interface be given?

6.3. Normal scopes and pseudo-scopes

Most scopes are *normal scopes*. The context object for a normal scope type is a mapping from each contextual type with that scope to an instance of that contextual type. There may be no more than one mapped instance per contextual type per thread. The set of all mapped instances of contextual types with a certain scope for a certain thread is called the *context* for that scope associated with that thread.

A context may be associated with one or more threads. A context with a certain scope is said to *propagate* from one point in the execution of the program to another when the set of mapped instances of contextual types with that scope is preserved.

The context associated with the current thread is called the *current context* for the scope. The mapped instance of a contextual type associated with a current context is called the *current instance* of the contextual type.

The `get()` operation of the context object for an active normal scope returns the current instance of the given contextual type.

At certain points in the execution of the program a context may be *destroyed*. When a context is destroyed, all mapped instances belonging to that context are destroyed by passing them to the `Contextual.destroy()` method.

Contexts with normal scopes must obey the following rule:

Suppose beans `A`, `B` and `Z` all have normal scopes. Suppose `A` has an injection point `x`, and `B` has an injection point `y`. Suppose further that both `x` and `y` resolve to bean `Z` according to the rules of typesafe resolution. If `a` is the current instance of `A`, and `b` is the current instance of `B`, then both `a.x` and `b.y` refer to the same instance of `Z`. This instance is the current instance of `Z`.

Any scope that is not a normal scope is called a *pseudo-scope*. The concept of a current instance is not well-defined in the case of a pseudo-scope.

All normal scopes must be explicitly declared `@NormalScope`, to indicate to the container that a client proxy is required.

All pseudo-scopes must be explicitly declared `@Scope`, to indicate to the container that no client proxy is required.

All scopes defined by this specification, except for the `@Dependent` pseudo-scope, are normal scopes.

6.4. Dependent pseudo-scope

The `@Dependent` scope type is a pseudo-scope. Beans declared with scope type `@Dependent` behave differently to beans with other built-in scope types.

When a bean is declared to have `@Dependent` scope:

- No injected instance of the bean is ever shared between multiple injection points.
- Any instance of the bean injected into an object that is being created by the container is bound to the lifecycle of the newly created object.
- When a Unified EL expression in a JSF or JSP page that refers to the bean by its bean name is evaluated, at most one instance of the bean is instantiated. This instance exists to service just a single evaluation of the EL expression. It is reused if the bean name appears multiple times in the EL expression, but is never reused when the EL expression is evaluated again, or when another EL expression is evaluated.
- Any instance of the bean that receives a producer method, producer field, disposer method or observer method invocation exists to service that invocation only.
- Any instance of the bean injected into method parameters of a disposer method or observer method exists to service the method invocation only (except for observer methods of container lifecycle events).

Every invocation of the `get()` operation of the `Context` object for the `@Dependent` scope with a `CreationalContext` returns a new instance of the given bean.

Every invocation of the `get()` operation of the `Context` object for the `@Dependent` scope with no `CreationalContext` returns a null value.

The `@Dependent` scope is always active.

6.4.1. Dependent objects

Many instances of beans with scope `@Dependent` belong to some other bean or Java EE component class instance and are called *dependent objects*.

- Instances of decorators and interceptors are dependent objects of the bean instance they decorate.
- An instance of a bean with scope `@Dependent` injected into a field, bean constructor or initializer method is a dependent object of the bean or Java EE component class instance into which it was injected.
- An instance of a bean with scope `@Dependent` injected into a producer method is a dependent object of the producer method bean instance that is being produced.
- An instance of a bean with scope `@Dependent` obtained by direct invocation of an `Instance` is a dependent object of the instance of `Instance`.

6.4.2. Destruction of objects with scope `@Dependent`

Dependent objects of a contextual instance are destroyed when `Contextual.destroy()` calls `CreationalContext.release()`, as defined in Section 6.1.1, “The CreationalContext interface”.

Additionally, the container must ensure that:

- all dependent objects of a non-contextual instance of a bean or other Java EE component class are destroyed when the instance is destroyed by the container,
- all `@Dependent` scoped contextual instances injected into method parameters of a disposer method or an observer method are destroyed when the invocation completes,
- any `@Dependent` scoped contextual instance created to receive a producer method, producer field, disposer method or observer method invocation is destroyed when the invocation completes, and
- all `@Dependent` scoped contextual instances created during evaluation of a Unified EL expression in a JSP or JSF page are destroyed when the evaluation completes.

Finally, the container is permitted to destroy any `@Dependent` scoped contextual instance at any time if the instance is no longer referenced by the application (excluding weak, soft and phantom references).

6.4.3. Dependent pseudo-scope and Unified EL

Suppose a Unified EL expression in a JSF or JSP page refers to a bean with scope `@Dependent` by its bean name. Each time the EL expression is evaluated:

- the bean is instantiated at most once, and
- the resulting instance is reused for every appearance of the bean name, and
- the resulting instance is destroyed when the evaluation completes.

Portable extensions that integrate with the container via Unified EL should also ensure that these rules are enforced.

6.5. Contextual instances and contextual references

The `Context` object is the ultimate source of the contextual instances that underly contextual references.

6.5.1. The active context object for a scope

From time to time, the container must obtain an *active context object* for a certain scope type. The container must search for an active instance of `Context` associated with the scope type.

- If no active context object exists for the scope type, the container throws a `ContextNotActiveException`.
- If more than one active context object exists for the given scope type, the container must throw an `IllegalStateException`.

If there is exactly one active instance of `Context` associated with the scope type, we say that the scope is *active*.

6.5.2. Contextual instance of a bean

From time to time, the container must obtain a *contextual instance* of a bean. The container must:

- obtain the active context object for the bean scope, then
- obtain an instance of the bean by calling `Context.get()`, passing the `Bean` instance representing the bean and an instance of `CreationalContext`.

From time to time, the container attempts to obtain a *contextual instance of a bean that already exists*, without creating a new contextual instance. The container must determine if the scope of the bean is active and if it is:

- obtain the active context object for the bean scope, then
- attempt to obtain an existing instance of the bean by calling `Context.get()`, passing the `Bean` instance representing the bean without passing any instance of `CreationalContext`.

If the scope is not active, or if `Context.get()` returns a null value, there is no contextual instance that already exists.

A contextual instance of any of the built-in kinds of bean defined in Chapter 3, *Programming model* is considered an internal container construct, and it is therefore not strictly required that a contextual instance of a built-in kind of bean directly implement the bean types of the bean. However, in this case, the container is required to transform its internal representation to an object that does implement the bean types expected by the application before injecting or returning a contextual instance to the application.

For a custom implementation of the `Bean` interface defined in Section 11.1, “The Bean interface”, the container calls `getScope()` to determine the bean scope.

6.5.3. Contextual reference for a bean

From time to time, the container must obtain a *contextual reference* for a bean and a given bean type of the bean. A contextual reference implements the given bean type and all bean types of the bean which are Java interfaces. A contextual reference is not, in general, required to implement all concrete bean types of the bean.

Contextual references must be obtained with a given `CreationalContext`, allowing any instance of scope `@Dependent` that is created to be later destroyed.

- If the bean has a normal scope and the given bean type cannot be proxied by the container, as defined in Section 3.15, “Unproxyable bean types”, the container throws an `UnproxyableResolutionException`.
- If the bean has a normal scope, then the contextual reference for the bean is a client proxy, as defined in Section 5.4, “Client proxies”, created by the container, that implements the given bean type and all bean types of the bean which are Java interfaces.
- Otherwise, if the bean has a pseudo-scope, the container must obtain a contextual instance of the bean. If the bean has scope `@Dependent`, the container must associate it with the `CreationalContext`.

The container must ensure that every injection point of type `InjectionPoint` and qualifier `@Default` of any dependent object instantiated during this process receives:

- an instance of `InjectionPoint` representing the injection point into which the dependent object will be injected, or
- a null value if it is not being injected into any injection point.

6.5.4. Contextual reference validity

A contextual reference for a bean is *valid* only for a certain period of time. The application should not invoke a method of an invalid reference.

The validity of a contextual reference for a bean depends upon whether the scope of the bean is a normal scope or a pseudo-scope.

- Any reference to a bean with a normal scope is valid as long as the application maintains a hard reference to it. However, it may only be invoked when the context associated with the normal scope is active. If it is invoked when the context is inactive, a `ContextNotActiveException` is thrown by the container.
- Any reference to a bean with a pseudo-scope (such as `@Dependent`) is valid until the bean instance to which it refers is destroyed. It may be invoked even if the context associated with the pseudo-scope is not active. If the application invokes a method of a reference to an instance that has already been destroyed, the behavior is undefined.

6.5.5. Injectable references

From time to time, the container must obtain an *injectable reference* for an injection point. The container must:

- Identify a bean according to the rules defined in Section 5.2, “Typesafe resolution” and resolving ambiguities according to Section 5.2.2, “Unsatisfied and ambiguous dependencies”.
- Obtain a contextual reference for this bean and the type of the injection point according to Section 6.5.3, “Contextual reference for a bean”.

For certain combinations of scopes, the container is permitted to optimize the above procedure:

- The container is permitted to directly inject a contextual instance of the bean, as defined in Section 6.5.2, “Contextual instance of a bean”.
- If an incompletely initialized instance of the bean is registered with the current `CreationalContext`, as defined in Section 6.1, “The Contextual interface”, the container is permitted to directly inject this instance.

However, in performing these optimizations, the container must respect the rules of *injectable reference validity*.

6.5.6. Injectable reference validity

Injectable references to a bean must respect the rules of contextual reference validity, with the following exceptions:

- A reference to a bean injected into a field, bean constructor or initializer method is only valid until the object into which it was injected is destroyed.
- A reference to a bean injected into a producer method is only valid until the producer method bean instance that is being produced is destroyed.
- A reference to a bean injected into a disposer method or observer method is only valid until the invocation of the method completes.

The application should not invoke a method of an invalid injected reference. If the application invokes a method of an invalid injected reference, the behavior is undefined.

6.6. Passivation and passivating scopes

The temporary transfer of the state of an idle object held in memory to some form of secondary storage is called *passivation*. The transfer of the passivated state back into memory is called *activation*.

6.6.1. Passivation capable beans

A bean is called *passivation capable* if the container is able to temporarily transfer the state of any idle instance to secondary storage.

- As defined by the EJB specification, all stateful session beans are passivation capable if and only if the all interceptors and decorators of the bean are passivation capable. Stateless and singleton session beans are not passivation capable.
- A managed bean is passivation capable if and only if the bean class is serializable and all interceptors and decorators of the bean are passivation capable.
- A producer method is passivation capable if and only if it never returns a value which is not passivation capable at runtime.
- A producer field is passivation capable if and only if it never refers to a value which is not passivation capable at runtime.

A custom implementation of `Bean` is passivation capable if it implements the interface `PassivationCapable`. An implementation of `Contextual` that is not a bean is passivation capable if it implements both `PassivationCapable` and `Serializable`.

```
public interface PassivationCapable {
    public String getId();
}
```

The `getId()` method must return a value that uniquely identifies the instance of `Bean` or `Contextual`. It is recommended that the string contain the package name of the class that implements `Bean` or `Contextual`.

6.6.2. Passivation capable dependencies

A bean is called a *passivation capable dependency* if any contextual reference for that bean is preserved when the object holding the reference is passivated and then activated.

The container must guarantee that:

- all beans with normal scope are passivation capable dependencies,
- all passivation capable beans with scope `@Dependent` are passivation capable dependencies,
- all resources are passivation capable dependencies, and
- the built-in beans of type `Instance`, `Event`, `InjectionPoint` and `BeanManager` are passivation capable dependencies.

A custom implementation of `Bean` is a passivation capable dependency if it implements `PassivationCapable` or if `getScope()` returns a normal scope type.

6.6.3. Passivating scopes

A *passivating scope* requires that:

- beans with the scope are passivation capable, and
- implementations of `Contextual` passed to any context object for the scope are passivation capable.

Passivating scopes must be explicitly declared `@NormalScope(passivating=true)`.

For example, the built-in session and conversation scopes defined in Section 6.7, “Context management for built-in scopes” are passivating scopes. No other built-in scopes are passivating scopes.

6.6.4. Validation of passivation capable beans and dependencies

For every bean which declares a passivating scope, the container must validate that the bean truly is passivation capable and that, in addition, its dependencies are passivation capable.

If a managed bean or stateful session bean which declares a passivating scope:

- is not passivation capable,
- has a non-transient injected field that does not resolve to a passivation capable dependency, or
- has an interceptor or decorator with a non-transient injected field that does not resolve to a passivation capable dependency,

then the container automatically detects the problem and treats it as a deployment problem.

If a producer method declares a passivating scope and has a return type that is declared final and does not implement or extend `Serializable` then the container automatically detects the problem and treats it as a deployment problem.

If a producer method declares a passivating scope and doesn't only return `Serializable` types at runtime, then the container must throw an `IllegalProductException`.

If a producer field declares a passivating scope and has a type that is declared final and does not implement or extend `Serializable` then the container automatically detects the problem and treats it as a deployment problem.

If a producer field declares a passivating scope and doesn't only contain `Serializable` values at runtime then the container must throw an `IllegalProductException`.

If a producer method or field of scope `@Dependent` returns an unserializable object for injection into an injection point that requires a passivation capable dependency, the container must throw an `IllegalProductException`

For a custom implementation of `Bean`, the container calls `getInjectionPoints()` to determine the injection points, and `InjectionPoint.isTransient()` to determine whether the injection point is a transient field.

If a bean which declares a passivating scope type, or any stateful session bean, has a decorator or interceptor which is not a passivation capable dependency, the container automatically detects the problem and treats it as a deployment problem.

6.7. Context management for built-in scopes

The container provides an implementation of the `Context` interface for each of the built-in scopes.

The built-in request and application context objects are active during servlet, web service and EJB invocations, and the built in session and request context objects are active during servlet and web service invocations. For other kinds of invocations, a portable extension may define a custom context object for any or all of the built-in scopes. For example, a remoting framework might provide a request context object for the built-in request scope.

The context associated with a built-in normal scope propagates across local, synchronous Java method calls, including invocation of EJB local business methods. The context does not propagate across remote method invocations or to asynchronous processes such as JMS message listeners or EJB timer service timeouts.

Portable extensions are encouraged to fire an event with qualifier `@Initialized(X.class)` when a custom context is initialized, and an event with qualifier `@Destroyed(X.class)` when a custom context is destroyed, where X is the scope type associated with the context. A suitable event payload should be chosen.

6.7.1. Request context lifecycle

The *request context* is provided by a built-in context object for the built-in scope type `@RequestScoped`. The request scope is active:

- during the `service()` method of any servlet in the web application, during the `doFilter()` method of any servlet filter and when the container calls any `ServletRequestListener` or `AsyncListener`,

- during any Java EE web service invocation,
- during any remote method invocation of any EJB, during any asynchronous method invocation of any EJB, during any call to an EJB timeout method and during message delivery to any EJB message-driven bean, and
- during `@PostConstruct` callback of any bean.

The request context is destroyed:

- at the end of the servlet request, after the `service()` method, all `doFilter()` methods, and all `requestDestroyed()` and `onComplete()` notifications return,
- after the web service invocation completes,
- after the EJB remote method invocation, asynchronous method invocation, timeout or message delivery completes if it did not already exist when the invocation occurred, or
- after the `@PostConstruct` callback completes, if it did not already exist when the `@PostConstruct` callback occurred.

An event with qualifier `@Initialized(RequestScoped.class)` is fired when the request context is initialized and an event with qualifier `@Destroyed(RequestScoped.class)` when the request context is destroyed. The event payload is:

- the `ServletRequestEvent` if the context is initialized or destroyed due to a servlet request, or
- the `ServletRequestEvent` if the context is initialized or destroyed due to a web service invocation, or
- any `java.lang.Object` for other types of request.

6.7.2. Session context lifecycle

The *session context* is provided by a built-in context object for the built-in passivating scope type `@SessionScoped`. The session scope is active:

- during the `service()` method of any servlet in the web application, during the `doFilter()` method of any servlet filter and when the container calls any `HttpSessionListener`, `AsyncListener` or `ServletRequestListener`.

The session context is shared between all servlet requests that occur in the same HTTP session. The session context is destroyed when the `HTTPSession` times out, after all `HttpSessionListeners` have been called, and at the very end of any request in which `invalidate()` was called, after all filters and `ServletRequestListeners` have been called.

An event with the `HttpSessionEvent` as payload and with qualifier `@Initialized(SessionScoped.class)` is fired when the session context is initialized and an event with qualifier `@Destroyed(SessionScoped.class)` when the session context is destroyed.

6.7.3. Application context lifecycle

The *application context* is provided by a built-in context object for the built-in scope type `@ApplicationScoped`. The application scope is active:

- during the `service()` method of any servlet in the web application, during the `doFilter()` method of any servlet filter and when the container calls any `ServletContextListener`, `HttpSessionListener`, `AsyncListener` or `ServletRequestListener`,
- during any Java EE web service invocation,
- during any remote method invocation of any EJB, during any asynchronous method invocation of any EJB, during any call to an EJB timeout method and during message delivery to any EJB message-driven bean,
- when the disposer method or `@PreDestroy` callback of any bean with any normal scope other than `@ApplicationScoped` is called, and

- during `@PostConstruct` callback of any bean.

The application context is shared between all servlet requests, web service invocations, EJB remote method invocations, EJB asynchronous method invocations, EJB timeouts and message deliveries to message-driven beans that execute within the same application. The application context is destroyed when the application is shut down.

An event with qualifier `@Initialized(ApplicationScoped.class)` is fired when the application context is initialized and an event with qualifier `@Destroyed(ApplicationScoped.class)` is fired when the application is destroyed. The event payload is:

- the `ServletContextEvent` if the application is a web application deployed to a Servlet container, or
- any `java.lang.Object` for other types of application.

6.7.4. Conversation context lifecycle

The *conversation context* is provided by a built-in context object for the built-in passivating scope type `@ConversationScoped`. The conversation scope is active during all Servlet requests.

An event with qualifier `@Initialized(ConversationScoped.class)` is fired when the conversation context is initialized and an event with qualifier `@Destroyed(ConversationScoped.class)` is fired when the conversation is destroyed. The event payload is:

- the conversation id if the conversation context is destroyed and is not associated with a current Servlet request, or
- the `ServletRequestEvent` if the application is a web application deployed to a Servlet container, or
- any `java.lang.Object` for other types of application.

The conversation context provides access to state associated with a particular *conversation*. Every Servlet request has an associated conversation. This association is managed automatically by the container according to the following rules:

- Any Servlet request has exactly one associated conversation.
- The conversation associated with a Servlet request is determined at the beginning of the request before calling any `service()` method of any servlet in the web application, calling the `doFilter()` method of any servlet filter in the web application and before the container calls any `ServletRequestListener` or `AsyncListener` in the web application.

Any conversation is in one of two states: *transient* or *long-running*.

- By default, a conversation is transient
- A transient conversation may be marked long-running by calling `Conversation.begin()`
- A long-running conversation may be marked transient by calling `Conversation.end()`

All long-running conversations have a string-valued unique identifier, which may be set by the application when the conversation is marked long-running, or generated by the container.

If the conversation associated with the current Servlet request is in the *transient* state at the end of a Servlet request, it is destroyed, and the conversation context is also destroyed.

If the conversation associated with the current Servlet request is in the *long-running* state at the end of a Servlet request, it is not destroyed. The long-running conversation associated with a request may be propagated to any Servlet request via use of a request parameter named `cid` containing the unique identifier of the conversation. In this case, the application must manage this request parameter.

If the current Servlet request is a JSF request, and the conversation is in *long-running* state, it is propagated according to the following rules:

- The long-running conversation context associated with a request that renders a JSF view is automatically propagated to any faces request (JSF form submission) that originates from that rendered page.

- The long-running conversation context associated with a request that results in a JSF redirect (a redirect resulting from a navigation rule or JSF `NavigationHandler`) is automatically propagated to the resulting non-faces request, and to any other subsequent request to the same URL. This is accomplished via use of a request parameter named `cid` containing the unique identifier of the conversation.

When no conversation is propagated to a Servlet request, or if a request parameter named `conversationPropagation` has the value `none` the request is associated with a new transient conversation.

All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

In the following cases, a propagated long-running conversation cannot be restored and reassociated with the request:

- When the HTTP servlet session is invalidated, all long-running conversation contexts created during the current session are destroyed, after the servlet `service()` method completes.
- The container is permitted to arbitrarily destroy any long-running conversation that is associated with no current Servlet request, in order to conserve resources.

The *conversation timeout*, which may be specified by calling `Conversation.setTimeout()` is a hint to the container that a conversation should not be destroyed if it has been active within the last given interval in milliseconds.

If the propagated conversation cannot be restored, the container must associate the request with a new transient conversation and throw an exception of type `javax.enterprise.context.NonexistentConversationException`.

The container ensures that a long-running conversation may be associated with at most one request at a time, by blocking or rejecting concurrent requests. If the container rejects a request, it must associate the request with a new transient conversation and throw an exception of type `javax.enterprise.context.BusyConversationException`.

6.7.5. The `Conversation` interface

The container provides a built-in bean with bean type `Conversation`, scope `@RequestScoped`, and qualifier `@Default`, named `javax.enterprise.context.conversation`.

```
public interface Conversation {
   public void begin();
   public void begin(String id);
   public void end();
   public String getId();
   public long getTimeout();
   public void setTimeout(long milliseconds);
   public boolean isTransient();
}
```

- `begin()` marks the current transient conversation long-running. A conversation identifier may, optionally, be specified. If no conversation identifier is specified, an identifier is generated by the container.
- `end()` marks the current long-running conversation transient.
- `getId()` returns the identifier of the current long-running conversation, or a null value if the current conversation is transient.
- `getTimeout()` returns the timeout, in milliseconds, of the current conversation.
- `setTimeout()` sets the timeout of the current conversation.
- `isTransient()` returns `true` if the conversation is marked transient, or `false` if it is marked long-running.

If any method of `Conversation` is called when the conversation scope is not active, a `ContextNotActiveException` is thrown.

If `end()` is called, and the current conversation is marked transient, an `IllegalStateException` is thrown.

If `begin()` is called, and the current conversation is already marked long-running, an `IllegalStateException` is thrown.

If `begin()` is called with an explicit conversation identifier, and a long-running conversation with that identifier already exists, an `IllegalArgumentException` is thrown.

OPEN ISSUE: The CDI 1.0 spec doesn't define when a context is "initialized" (however the concept is implied), however the CDI spec introduces lifecycle events that require an event to be fired when a context is initialized. Either we need to re-write the definition of this feature to use the CDI 1.0 language or we need to define when a context is initialized.

OPEN ISSUE: What should the event type be for applications initialized or destroyed when the application is not a web app?

Chapter 7. Lifecycle of contextual instances

The lifecycle of a contextual instance of a bean is managed by the context object for the bean's scope, as defined in Chapter 6, *Scopes and contexts*.

Every bean in the system is represented by an instance of the `Bean` interface defined in Section 11.1, “The Bean interface”. This interface is a subtype of `Contextual`. To create and destroy contextual instances, the context object calls the `create()` and `destroy()` operations defined by the interface `Contextual`, as defined in Section 6.1, “The Contextual interface”.

7.1. Restriction upon bean instantiation

The managed bean and EJB specifications place very few programming restrictions upon the bean class of a bean. In particular, the class is a concrete class and is not required to implement any special interface or extend any special superclass. Therefore, bean classes are easy to instantiate and unit test.

However, if the application directly instantiates a bean class, instead of letting the container perform instantiation, the resulting instance is not managed by the container and is not a contextual instance as defined by Section 6.5.2, “Contextual instance of a bean”. Furthermore, the capabilities listed in Section 2.1, “Functionality provided by the container to the bean” will not be available to that particular instance. In a deployed application, it is the container that is responsible for instantiating beans and initializing their dependencies.

If the application requires more control over instantiation of a contextual instance, a producer method or field may be used. Any Java object may be returned by a producer method or field. It is not required that the returned object be a contextual reference for a bean. However, if the object is not a contextual reference for another bean, the object will be contextual instance of the producer method bean, and therefore available for injection into other objects and use in EL expressions, but the other capabilities listed in Section 2.1, “Functionality provided by the container to the bean” will not be available to the object.

In the following example, a producer method returns instances of other beans:

```
@SessionScoped
public class PaymentStrategyProducer implements Serializable {

   private PaymentStrategyType paymentStrategyType;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      paymentStrategyType = type;
   }

   @Produces PaymentStrategy getPaymentStrategy(@CreditCard PaymentStrategy creditCard,
                                                @Cheque PaymentStrategy cheque,
                                                @Online PaymentStrategy online) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return creditCard;
         case CHEQUE: return cheque;
         case ONLINE: return online;
         default: throw new IllegalStateException();
      }
    }

}
```

In this case, any object returned by the producer method has already had its dependencies injected, receives lifecycle callbacks and event notifications and may have interceptors.

But in this example, the returned objects are not contextual instances:

```
@SessionScoped
public class PaymentStrategyProducer implements Serializable {

   private PaymentStrategyType paymentStrategyType;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      paymentStrategyType = type;
   }

   @Produces PaymentStrategy getPaymentStrategy() {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return new CreditCardPaymentStrategy();
         case CHEQUE: return new ChequePaymentStrategy();
```

```
        case ONLINE: return new OnlinePaymentStrategy();
        default: throw new IllegalStateException();
      }
    }

}
```

In this case, any object returned by the producer method will not have any dependencies injected by the container, receives no lifecycle callbacks or event notifications and does not have interceptors or decorators.

7.2. Container invocations and interception

When the application invokes:

- a method of a bean via a contextual reference to the bean, as defined in Section 6.5.3, “Contextual reference for a bean”, or
- a business method of a session bean via an EJB remote or local reference,

the invocation is treated as a *business method invocation*.

When the container invokes a method of a bean, the invocation may or may not be treated as a business method invocation:

- Invocations of initializer methods by the container are not business method invocations.
- Invocations of producer, disposer and observer methods by the container are business method invocations and are intercepted by method interceptors and decorators.
- Invocation of lifecycle callbacks by the container are not business method invocations, but are intercepted by interceptors for lifecycle callbacks.
- Invocation of EJB timer service timeouts by the container are not business method invocations, but are intercepted by interceptors for EJB timeouts.
- Invocations of interceptors and decorator methods during method or lifecycle callback interception are not business method invocations, and therefore no recursive interception occurs.
- Invocations of message listener methods of message-driven beans during message delivery are business method invocations.
- Invocations of methods declared by java.lang.Object are not business method invocations.

If, and only if, an invocation is a business method invocation:

- it passes through method interceptors and decorators, and
- in the case of a session bean, it is subject to EJB services such as declarative transaction management, concurrency, security and asynchronicity, as defined by the EJB specification.

Otherwise, the invocation is treated as a normal Java method call and is not intercepted by the container.

7.3. Lifecycle of contextual instances

The actual mechanics of bean creation and destruction varies according to what kind of bean is being created or destroyed.

7.3.1. Lifecycle of managed beans

When the `create()` method of the `Bean` object that represents a managed bean is called, the container obtains an instance of the bean, as defined by the Managed Beans specification, calling the bean constructor as defined by Section 5.5.1, “Injection using the bean constructor”, and performing dependency injection as defined in Section 5.5.2, “Injection of fields and initializer methods”.

When the `destroy()` method is called, the container destroys the instance, as defined by the Managed Beans specification, and any dependent objects, as defined in Section 5.5.3, “Destruction of dependent objects”.

7.3.2. Lifecycle of stateful session beans

When the `create()` method of a `Bean` object that represents a stateful session bean that is called, the container creates and returns a container-specific internal local reference to a new session bean instance. The reference must be passivation capable. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying stateful session bean instance. This object must be passivation capable.

When the `destroy()` method is called, and if the underlying EJB was not already removed by direct invocation of a remove method by the application, the container removes the stateful session bean. The `@PreDestroy` callback must be invoked by the container.

Note that the container performs additional work when the underlying EJB is created and removed, as defined in Section 5.5, “Dependency injection”

7.3.3. Lifecycle of stateless session and singleton beans

When the `create()` method of a `Bean` object that represents a stateless session or singleton session bean is called, the container creates and returns a container-specific internal local reference to the session bean. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying session bean. This object must be passivation capable.

When the `destroy()` method is called, the container simply discards this internal reference.

Note that the container performs additional work when the underlying EJB is created and removed, as defined in Section 5.5, “Dependency injection”

7.3.4. Lifecycle of producer methods

When the `create()` method of a `Bean` object that represents a producer method is called, the container must invoke the producer method as defined by Section 5.5.4, “Invocation of producer or disposer methods”. The return value of the producer method, after method interception completes, is the new contextual instance to be returned by `Bean.create()`.

If the producer method returns a null value and the producer method bean has the scope `@Dependent`, the `create()` method returns a null value.

Otherwise, if the producer method returns a null value, and the scope of the producer method is not `@Dependent`, the `create()` method throws an `IllegalProductException`.

When the `destroy()` method is called, and if there is a disposer method for this producer method, the container must invoke the disposer method as defined by Section 5.5.4, “Invocation of producer or disposer methods”, passing the instance given to `destroy()` to the disposed parameter. Finally, the container destroys dependent objects, as defined in Section 5.5.3, “Destruction of dependent objects”.

7.3.5. Lifecycle of producer fields

When the `create()` method of a `Bean` object that represents a producer field is called, the container must access the producer field as defined by Section 5.5.5, “Access to producer field values” to obtain the current value of the field. The value of the producer field is the new contextual instance to be returned by `Bean.create()`.

If the producer field contains a null value and the producer field bean has the scope `@Dependent`, the `create()` method returns a null value.

Otherwise, if the producer field contains a null value, and the scope of the producer field is not `@Dependent`, the `create()`

method throws an `IllegalProductException`.

When the `destroy()` method is called, and if there is a disposer method for this producer field, the container must invoke the disposer method as defined by Section 5.5.4, “Invocation of producer or disposer methods”, passing the instance given to `destroy()` to the disposed parameter.

7.3.6. Lifecycle of resources

When the `create()` method of a `Bean` object that represents a resource is called, the container creates and returns a container-specific internal reference to the Java EE component environment resource, entity manager, entity manager factory, remote EJB instance or web service reference. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying resource, entity manager, entity manager factory, remote EJB instance or web service reference. This object must be passivation capable.

The container must perform ordinary Java EE component environment injection upon any non-static field that functions as a resource declaration, as defined by the Java EE platform and Common Annotations for the Java platform specifications. The container is not required to perform Java EE component environment injection upon a static field. Portable applications should not rely upon the value of a static field that functions as a resource declaration.

References to EJBs and web services are always dependent scoped and a new instance must be obtained for every injection performed.

For an entity manager associated with a resource definition, it must behave as though it were injected directly using `@PersistencContext`.

When the `destroy()` method of a bean which represents a remote stateful EJB reference is called, the container will *not* automatically destroy the EJB reference. The application must explicitly call the method annotated `@Remove`. This behavior differs to that specified in Section 7.3.2, “Lifecycle of stateful session beans” for beans which represent a local stateful EJB reference

Chapter 8. Decorators

A *decorator* implements one or more bean types and intercepts business method invocations of beans which implement those bean types. These bean types are called *decorated types*.

Decorators are superficially similar to interceptors, but because they directly implement operations with business semantics, they are able to implement business logic and, conversely, unable to implement the cross-cutting concerns for which interceptors are optimized.

Decorators may be associated with any managed bean that is not itself an interceptor or decorator, with any EJB session bean or with any built-in bean. Decorators are not applied to the return value of a producer method or the current value of a producer field. A decorator instance is a dependent object of the object it decorates.

8.1. Decorator beans

A decorator is a managed bean. The set of decorated types of a decorator includes all bean types of the managed bean which are Java interfaces, except for `java.io.Serializable`. The decorator bean class and its superclasses are not decorated types of the decorator. The decorator class may be abstract.

If the set of decorated types of a decorator is empty, the container automatically detects the problem and treats it as a definition error.

Decorators of a session bean must comply with the bean provider programming restrictions defined by the EJB specification. Decorators of a stateful session bean must comply with the rules for instance passivation and conversational state defined by the EJB specification.

8.1.1. Declaring a decorator

A decorator is declared by annotating the bean class with the `@javax.decorator.Decorator` stereotype.

```
@Decorator
class TimestampLogger implements Logger { ... }
```

8.1.2. Decorator delegate injection points

All decorators have a *delegate injection point*. A delegate injection point is an injection point of the bean class. The type and qualifiers of the injection point are called the *delegate type* and *delegate qualifiers*. The decorator applies to beans that are assignable to the delegate injection point.

The delegate injection point must be declared by annotating the injection point with the annotation `@javax.decorator.Delegate`:

```
@Decorator
class TimestampLogger implements Logger {
    @Inject @Delegate @Any Logger logger;
    ...
}
```

```
@Decorator
class TimestampLogger implements Logger {
    private Logger logger;

    @Inject
    public TimestampLogger(@Delegate @Debug Logger logger) {
        this.logger=logger;
    }
    ...
}
```

A decorator must have exactly one delegate injection point. If a decorator has more than one delegate injection point, or does not have a delegate injection point, the container automatically detects the problem and treats it as a definition error.

The delegate injection point must be an injected field, initializer method parameter or bean constructor method parameter. If an injection point that is not an injected field, initializer method parameter or bean constructor method parameter is annotated `@Delegate`, the container automatically detects the problem and treats it as a definition error.

If a bean class that is not a decorator has an injection point annotated `@Delegate`, the container automatically detects the problem and treats it as a definition error.

The container must inject a *delegate* object to the delegate injection point. The delegate object implements the delegate type and delegates method invocations to remaining uninvoked decorators and eventually to the bean. When the container calls a decorator during business method interception, the decorator may invoke any method of the delegate object.

```
@Decorator
class TimestampLogger implements Logger {
    @Inject @Delegate @Any Logger logger;

    void log(String message) {
        logger.log( timestamp() + ": " + message );
    }
    ...
}
```

If a decorator invokes the delegate object at any other time, the invoked method throws an `IllegalStateException`.

8.1.3. Decorated types of a decorator

The delegate type of a decorator must implement or extend every decorated type (with exactly the same type parameters). If the delegate type does not implement or extend a decorated type of the decorator (or specifies different type parameters), the container automatically detects the problem and treats it as a definition error.

A decorator is not required to implement the delegate type.

A decorator may be an abstract Java class, and is not required to implement every method of every decorated type. Whenever the decorator does not implement a method of the decorated type, the container will provide an implicit implementation that calls the method on the delegate. If a decorator has abstract methods that are not declared by a decorated type, the container automatically detects the problem and treats it as a definition error.

The decorator intercepts every method which is declared by a decorated type of the decorator and is implemented by the bean class of the decorator.

8.2. Decorator enablement and ordering

Decorators may be enabled for an application, or enabled only for a particular bean archive.

A decorator may be enabled for the entire application by listing its class, along with a priority attribute, under the `<decorators>` element of the `beans.xml` file of the bean archive which contains the decorator class.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
      <class priority="100">org.mycompany.myfwk.TimestampLogger</class>
      <class priority="200">org.mycompany.myfwk.IdentityLogger</class>
   </decorators>
</beans>
```

A decorator may be enabled for a bean archive by listing its class under the `<decorators>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
      <class>org.mycompany.myfwk.OrderingDecorator</class>
   </decorators>
</beans>
```

A decorator enabled for an application may be disabled for a bean archive by listing its class, along with an enabled flag, under the `<decorators>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
```

```
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
      <class enabled="false">org.mycompany.myfwk.TimestampLogger</class>
   </decorators>
</beans>
```

A decorator may be given a default priority, but not enabled by listing its class, along with a disabled flag and the priority attribute, under the `<decorators>` element of the `beans.xml` file of the bean archive which contains the decorator class.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
      <class enabled="false" priority="100">org.mycompany.myfwk.TimestampLogger</class>
   </decorators>
</beans>
```

A decorator with a default priority may be enabled for a bean archive by listing its class, along with an enabled flag, under the `<decorators>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
      <class enabled="true">org.mycompany.myfwk.TimestampLogger</class>
   </decorators>
</beans>
```

OPEN ISSUE: Can other libraries disable decorators and change the priority? If so, which one wins?

OPEN ISSUE: If a library defines an enabled or default priority decorator, can it be disabled or enabled for the entire application?

OPEN ISSUE: Should we add a way to specify the priority by annotation?

The priority of the decorator declaration, defined in Section 2.9, “Priority ordering”, determines the default decorator ordering for the application. The order of the decorator declarations for a bean archive overrides the default decorator order. If the set of decorators enabled for the application and the set of decorators enabled for a bean archive are disjoint, then the first decorator enabled for the bean archive is given a priority of 1000. Decorators which occur earlier in the list are called first.

By default, a bean archive has no enabled decorators. A decorator must be explicitly enabled by listing its bean class under the `<decorators>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
      <class>org.mycompany.myfwk.TimestampLogger</class>
      <class>org.mycompany.myfwk.IdentityLogger</class>
   </decorators>
</beans>
```

The order of the decorator declarations determines the decorator ordering. Decorators which occur earlier in the list are called first.

Each child `<class>` element must specify the name of a decorator bean class. If there is no class with the specified name, or if the class with the specified name is not a decorator bean class, the container automatically detects the problem and treats it as a deployment problem.

If the same class is listed twice under the `<decorators>` element, the container automatically detects the problem and treats it as a deployment problem.

Decorators are called after interceptors.

A decorator is said to be *enabled* if it is enabled in at least one bean archive.

8.3. Decorator resolution

The process of matching decorators to a certain bean is called *decorator resolution*. A decorator is bound to a bean if:

- The bean is assignable to the delegate injection point according to the rules defined in Section 5.2, "Typesafe resolution" (using Section 8.3.1, "Assignability of raw and parameterized types for delegate injection points").
- The decorator is enabled in the bean archive containing the bean.

If a decorator matches a managed bean, and the managed bean class is declared final, the container automatically detects the problem and treats it as a deployment problem.

If a decorator matches a managed bean with a non-static, non-private, final method, and the decorator also implements that method, the container automatically detects the problem and treats it as a deployment problem.

For a custom implementation of the `Decorator` interface defined in Section 11.1.1, "The Decorator interface", the container calls `getDelegateType()`, `getDelegateQualifiers()` and `getDecoratedTypes()` to determine the delegate type and qualifiers and decorated types of the decorator.

8.3.1. Assignability of raw and parameterized types for delegate injection points

Decorator delegate injection points have a special set of rules for determining assignability of raw and parameterized types, as an exception to Section 5.2.4, "Assignability of raw and parameterized types".

A raw bean type is considered assignable to a parameterized delegate type if the raw types are identical and all type parameters of the delegate type are either unbounded type variables or `java.lang.Object`.

A parameterized bean type is considered assignable to a parameterized delegate type if they have identical raw type and for each parameter:

- the delegate type parameter and the bean type parameter are actual types with identical raw type, and, if the type is parameterized, the bean type parameter is assignable to the delegate type parameter according to these rules, or
- the delegate type parameter is a wildcard, the bean type parameter is an actual type and the actual type is assignable to the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the delegate type parameter is a wildcard, the bean type parameter is a type variable and the upper bound of the type variable is assignable to the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the delegate type parameter and the bean type parameter are both type variables and the upper bound of the bean type parameter is assignable to the upper bound, if any, of the delegate type parameter, or
- the delegate type parameter is a type variable, the bean type parameter is an actual type, and the actual type is assignable to the upper bound, if any, of the type variable.

8.4. Decorator invocation

Whenever a business method is invoked on an instance of a bean with decorators, the container intercepts the business method invocation and, after processing all interceptors of the method, invokes decorators of the bean.

The container searches for the first decorator of the instance that implements the method that is being invoked as a business method. If no such decorator exists, the container invokes the business method of the intercepted instance. Otherwise, the container calls the method of the decorator.

When any decorator is invoked by the container, it may in turn invoke a method of the delegate. The container intercepts the delegate invocation and searches for the first decorator of the instance such that:

- the decorator occurs after the decorator invoking the delegate, and
- the decorator implements the method that is being invoked upon the delegate.

If no such decorator exists, the container invokes the business method of the intercepted instance. Otherwise, the container calls the method of the decorator.

Chapter 9. Interceptor bindings

Managed beans and EJB session and message-driven beans support interception. *Interceptors* are used to separate cross-cutting concerns from business logic. The Java Interceptors specification defines the basic programming model and semantics. This specification defines a typesafe mechanism for associating interceptors to beans using *interceptor bindings*.

Interceptor bindings may be used to associate interceptors with any managed bean that is not itself an interceptor or decorator or with any EJB session or message-driven bean. Interceptors are not associated with the return value of a producer method or the current value of a producer field. An interceptor instance is a dependent object of the object it intercepts.

9.1. Interceptor binding types

An *interceptor binding type* is a Java annotation defined as `@Retention(RUNTIME)`. Typically an interceptor binding is defined as `@Target({TYPE, METHOD})` or `@Target(TYPE)`.

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

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

9.1.1. Interceptor binding types with additional interceptor bindings

An interceptor binding type may declare other interceptor bindings.

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

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

Interceptor binding types declared `@Target(TYPE)` may not be applied to interceptor binding types declared `@Target({TYPE, METHOD})`.

9.1.2. Interceptor bindings for stereotypes

Interceptor bindings may be applied to a stereotype by annotating the stereotype annotation:

```
@Transactional
@Secure
@RequestScoped
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

An interceptor binding declared by a stereotype is inherited by any bean that declares that stereotype.

If a stereotype declares interceptor bindings, it must be defined as `@Target(TYPE)`.

9.2. Declaring the interceptor bindings of an interceptor

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

```
@Transactional @Interceptor
public class TransactionInterceptor {
```

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

}
```

An interceptor class may declare multiple interceptor bindings.

Multiple interceptors may declare the same interceptor bindings.

If an interceptor does not declare an `@Interceptor` annotation, it must be bound to beans using `@Interceptors` or `ejb-jar.xml`.

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

An interceptor for lifecycle callbacks may only declare interceptor binding types that are defined as `@Target(TYPE)`. If an interceptor for lifecycle callbacks declares an interceptor binding type that not defined `@Target(TYPE)`, the container automatically detects the problem and treats it as a definition error.

9.3. Binding an interceptor to a bean

An interceptor binding may be declared by annotating the bean class, or a method of the bean class, with the interceptor binding type.

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

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

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

```
public class ShoppingCart {

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

}
```

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

The set of interceptor bindings for a method are those declared at class level, including those declared on stereotypes, combined with those declared at method level.

- An interceptor binding declared on a bean class replaces an interceptor binding of the same type declared by a stereotype that is applied to the bean class.
- An interceptor binding declared on a method replaces an interceptor binding of the same type declared, or inherited, at class level, or an interceptor binding of the same type declared by a stereotype that is applied to the bean class.

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

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

9.4. Interceptor enablement and ordering

Interceptors may be enabled for an application, or enabled only for a particular bean archive.

An interceptor may be enabled for the entire application by listing its class, along with a priority attribute, under the `<interceptors>` element of the `beans.xml` file of the bean archive which contains the interceptor class.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class priority="100">org.mycompany.myfwk.TransactionInterceptor</class>
      <class priority="200">org.mycompany.myfwk.LoggingInterceptor</class>
   </interceptors>
</beans>
```

An interceptor may be enabled for a bean archive by listing its class under the `<interceptors>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class>org.mycompany.myfwk.ValidationInterceptor</class>
      <class>org.mycompany.myfwk.SecurityInterceptor</class>
   </interceptors>
</beans>
```

An interceptor enabled for an application may be disabled for a bean archive by listing its class, along with a disabled flag, under the `<interceptors>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class enabled="false">org.mycompany.myfwk.TransactionInterceptor</class>
   </interceptors>
</beans>
```

An interceptor may be given a default priority, but not enabled by listing its class, along with a disabled flag and the priority attribute, under the `<interceptors>` element of the `beans.xml` file of the bean archive which contains the interceptor class.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class enabled="false" priority="100">org.mycompany.myfwk.TransactionInterceptor</class>
   </interceptors>
</beans>
```

An interceptor with a default priority may be enabled for a bean archive by listing its class, along with an enabled flag, under the `<interceptors>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class enabled="true">org.mycompany.myfwk.TransactionInterceptor</class>
   </interceptors>
</beans>
```

OPEN ISSUE: Can other libraries disable interceptors and change the priority? If so, which one wins?

OPEN ISSUE: If a library defines an enabled or default priority interceptor, can it be disabled or enabled for the entire application?

OPEN ISSUE: Should we add a way to specify the priority by annotation?

The priority of the interceptor declaration, defined in Section 2.9, “Priority ordering”, determines the default interceptor ordering for the application. The order of the interceptor declarations for a bean archive overrides the default interceptor order. If the set of interceptors enabled for the application and the set of interceptors enabled for a bean archive are disjoint, then the interceptors enabled for the bean archive are given an incrementing priority starting at 1000. Interceptors which occur earlier in the list are called first.

Each child `<class>` element must specify the name of an interceptor class. If there is no class with the specified name, or if the class with the specified name is not an interceptor class, the container automatically detects the problem and treats it as a deployment problem.

If the same class is listed twice under the `<interceptors>` element, the container automatically detects the problem and treats it as a deployment problem.

Interceptors declared using `@Interceptors` or in `ejb-jar.xml` are called before interceptors declared using interceptor bindings.

Interceptors are called before decorators.

An interceptor is said to be *enabled* if it is enabled in at least one bean archive.

9.5. Interceptor resolution

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

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

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

An interceptor is bound to a method if:

- The method has all the interceptor bindings of the interceptor. A method has an interceptor binding of an interceptor if it has an interceptor binding with (a) the same type and (b) the same annotation member value for each member which is not annotated `@javax.enterprise.util.Nonbinding`.
- The interceptor intercepts the given kind of lifecycle callback or business method.
- The interceptor is enabled in the bean archive containing the bean.

For a custom implementation of the `Interceptor` interface defined in Section 11.1.2, “The Interceptor interface”, the container calls `getInterceptorBindings()` to determine the interceptor bindings of the interceptor and `intercepts()` to determine if the interceptor intercepts a given kind of lifecycle callback, EJB timeout or business method.

9.5.1. Interceptors with multiple bindings

An interceptor class may specify multiple interceptor bindings.

```
@Transactional @Secure @Interceptor
public class TransactionalSecurityInterceptor {

   @AroundInvoke
   public Object aroundInvoke(InvocationContext context) throws Exception { ... }

}
```

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

```
@Transactional @Secure
public class ShoppingCart { ... }
```

The interceptor will also be bound to the `placeOrder()` method of this bean:

```
@Transactional
public class ShoppingCart {

   @Secure
```

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

}
```

However, it will not be bound to the `placeOrder()` method of this bean, since the `@Secure` interceptor binding does not appear:

```
@Transactional
public class ShoppingCart {

    public void placeOrder() { ... }

}
```

9.5.2. Interceptor binding types with members

Interceptor binding types may have annotation members.

```
@Inherited
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transactional {
    boolean requiresNew() default false;
}
```

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

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {

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

}
```

The `RequiresNewTransactionInterceptor` applies to this bean:

```
@Transactional(requiresNew=true)
public class ShoppingCart { ... }
```

But not to this bean:

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

Annotation member values are compared using `equals()`.

An annotation member may be excluded from consideration using the `@Nonbinding` annotation.

```
@Inherited
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transactional {
    @Nonbinding boolean requiresNew() default false;
}
```

Array-valued or annotation-valued members of an interceptor binding type should be annotated `@Nonbinding` in a portable application. If an array-valued or annotation-valued member of an interceptor binding type is not annotated `@Nonbinding`, non-portable behavior results.

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

Chapter 10. Events

Beans may produce and consume events. This facility allows beans to interact in a completely decoupled fashion, with no compile-time dependency between the interacting beans. Most importantly, it allows stateful beans in one architectural tier of the application to synchronize their internal state with state changes that occur in a different tier.

An event comprises:

- A Java object—the *event object*
- A set of instances of qualifier types—the *event qualifiers*

The event object acts as a payload, to propagate state from producer to consumer. The event qualifiers act as topic selectors, allowing the consumer to narrow the set of events it observes.

An *observer method* acts as event consumer, observing events of a specific type—the *observed event type*—with a specific set of qualifiers—the *observed event qualifiers*. An observer method will be notified of an event if the event object is assignable to the observed event type, and if all the observed event qualifiers are event qualifiers of the event.

10.1. Event types and qualifier types

An event object is an instance of a concrete Java class with no type variables. The *event types* of the event include all superclasses and interfaces of the runtime class of the event object.

An event type may not contain a type variable.

An event qualifier type is just an ordinary qualifier type as specified in Section 2.3.2, “Defining new qualifier types”, typically defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})` or `@Target({FIELD, PARAMETER})`.

Every event has the qualifier `@javax.enterprise.inject.Any`, even if it does not explicitly declare this qualifier.

Any Java type may be an observed event type.

10.2. Observer resolution

The process of matching an event to its observer methods is called *observer resolution*. The container considers event type and qualifiers when resolving observers.

Observer resolution usually occurs at runtime.

An event is delivered to an observer method if:

- The observer method belongs to an enabled bean.
- The event object is assignable to the observed event type, taking type parameters into consideration.
- The observer method has no event qualifiers or has a subset of the event qualifiers. An observer method has an event qualifier if it has an observed event qualifier with (a) the same type and (b) the same annotation member value for each member which is not annotated `@javax.enterprise.util.Nonbinding`.
- Either the event is not a container lifecycle event, as defined in Section 11.5, “Container lifecycle events”, or the observer method belongs to an extension.

If the runtime type of the event object contains a type variable, the container must throw an `IllegalArgumentException`.

For a custom implementation of the `ObserverMethod` interface defined in Section 11.1.3, “The ObserverMethod interface”, the container must call `getObservedType()` and `getObservedQualifiers()` to determine the observed event type and qualifiers.

10.2.1. Assignability of type variables, raw and parameterized types

An event type is considered assignable to a type variable if the event type is assignable to the upper bound, if any.

A parameterized event type is considered assignable to a raw observed event type if the raw types are identical.

A parameterized event type is considered assignable to a parameterized observed event type if they have identical raw type and for each parameter:

- the observed event type parameter is an actual type with identical raw type to the event type parameter, and, if the type is parameterized, the event type parameter is assignable to the observed event type parameter according to these rules, or
- the observed event type parameter is a wildcard and the event type parameter is assignable to the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the observed event type parameter is a type variable and the event type parameter is assignable to the upper bound, if any, of the type variable.

10.2.2. Event qualifier types with members

As usual, the qualifier type may have annotation members:

```
@Qualifier
@Target(PARAMETER)
@Retention(RUNTIME)
public @interface Role {
    String value();
}
```

Consider the following event:

```
public void login() {
    final User user = ...;
    loggedInEvent.fire( new LoggedInEvent(user),
            new RoleQualifier() { public String value() { return user.getRole(); } );
}
```

Where `RoleQualifier` is an implementation of the qualifier type `Role`:

```
public abstract class RoleQualifier
        extends AnnotationLiteral<Role>
        implements Role {}
```

Then the following observer method will always be notified of the event:

```
public void afterLogin(@Observes LoggedInEvent event) { ... }
```

Whereas this observer method may or may not be notified, depending upon the value of `user.getRole()`:

```
public void afterAdminLogin(@Observes @Role("admin") LoggedInEvent event) { ... }
```

As usual, the container uses `equals()` to compare event qualifier type member values.

10.2.3. Multiple event qualifiers

An event parameter may have multiple qualifiers.

```
public void afterDocumentUpdatedByAdmin(@Observes @Updated @ByAdmin Document doc) { ... }
```

Then this observer method will only be notified if all the observed event qualifiers are specified when the event is fired:

```
documentEvent.fire( document, new UpdatedQualifier() {}, new ByAdminQualifier() {} );
```

Other, less specific, observers will also be notified of this event:

```
public void afterDocumentUpdated(@Observes @Updated Document doc) { ... }
```

```
public void afterDocumentEvent(@Observes Document doc) { ... }
```

10.3. Firing events

Beans fire events via an instance of the `javax.enterprise.event.Event` interface, which may be injected:

```
@Inject @Any Event<LoggedInEvent> loggedInEvent;
```

The method `fire()` accepts an event object:

```
public void login() {
    ...
    loggedInEvent.fire( new LoggedInEvent(user) );
}
```

Any combination of qualifiers may be specified at the injection point:

```
@Inject @Admin Event<LoggedInEvent> adminLoggedInEvent;
```

Or, the `@Any` qualifier may be used, allowing the application to specify qualifiers dynamically:

```
@Inject @Any Event<LoggedInEvent> loggedInEvent;
...
LoggedInEvent event = new LoggedInEvent(user);
if ( user.isAdmin() ) {
    loggedInEvent.select( new AdminQualifier() ).fire(event);
}
else {
    loggedInEvent.fire(event);
}
```

In this example, the event sometimes has the qualifier `@Admin`, depending upon the value of `user.isAdmin()`.

10.3.1. The `Event` interface

The `Event` interface provides a method for firing events with a specified combination of type and qualifiers:

```
public interface Event<T> {

    public void fire(T event);

    public Event<T> select(Annotation... qualifiers);
    public <U extends T> Event<U> select(Class<U> subtype, Annotation... qualifiers);
    public <U extends T> Event<U> select(TypeLiteral<U> subtype, Annotation... qualifiers);

}
```

For an injected `Event`:

- the *specified type* is the type parameter specified at the injection point, and
- the *specified qualifiers* are the qualifiers specified at the injection point.

For example, this injected `Event` has specified type `LoggedInEvent` and specified qualifier `@Any`:

```
@Inject @Any Event<LoggedInEvent> any;
```

The `select()` method returns a child `Event` for a given specified type and additional specified qualifiers. If no specified type is given, the specified type is the same as the parent.

For example, this child `Event` has required type `AdminLoggedInEvent` and additional specified qualifier `@Admin`:

```
Event<AdminLoggedInEvent> admin = any.select(
            AdminLoggedInEvent.class,
            new AdminQualifier() );
```

If the specified type contains a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same qualifier type are passed to `select()`, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a qualifier type is passed to `select()`, an `IllegalArgumentException` is thrown.

The method `fire()` fires an event with the specified qualifiers and notifies observers, as defined by Section 10.5, “Observer notification”.

If the runtime type of the event object contains a type variable, an `IllegalArgumentException` is thrown.

If the runtime type of the event object is assignable to the type of a container lifecycle event, `IllegalArgumentException` is thrown.

10.3.2. The built-in `Event`

The container must provide a built-in bean with:

- `Event<X>` in its set of bean types, for every Java type `X` that does not contain a type variable,
- every event qualifier type in its set of qualifier types,
- scope `@Dependent`,
- no bean name, and
- an implementation provided automatically by the container.

If an injection point of raw type `Event` is defined, the container automatically detects the problem and treats it as a definition error.

The built-in implementation must be a passivation capable dependency, as defined in Section 6.6.2, “Passivation capable dependencies”.

10.4. Observer methods

An observer method allows the application to receive and respond to event notifications.

An observer method is a non-abstract method of a managed bean class or session bean class (or of an extension, as defined in Section 11.5, “Container lifecycle events”). An observer method may be either static or non-static. If the bean is a session bean, the observer method must be either a business method of the EJB or a static method of the bean class.

There may be arbitrarily many observer methods with the same event parameter type and qualifiers.

A bean (or extension) may declare multiple observer methods.

10.4.1. Event parameter of an observer method

Each observer method must have exactly one *event parameter*, of the same type as the event type it observes. When searching for observer methods for an event, the container considers the type and qualifiers of the event parameter.

If the event parameter does not explicitly declare any qualifier, the observer method observes events with no qualifier.

The event parameter type may contain a type variable or wildcard.

10.4.2. Declaring an observer method

An observer method may be declared by annotating a parameter `@javax.enterprise.event.Observes` of a default-access, public, protected or private method. That parameter is the event parameter. The declared type of the parameter is the observed event type.

```
public void afterLogin(@Observes LoggedInEvent event) { ... }
```

If a method has more than one parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error.

Observed event qualifiers may be declared by annotating the event parameter:

```
public void afterLogin(@Observes @Admin LoggedInEvent event) { ... }
```

If an observer method is annotated `@Produces` or `@Inject` or has a parameter annotated `@Disposes`, the container automatically detects the problem and treats it as a definition error.

If a non-static method of a session bean class has a parameter annotated `@Observes`, and the method is not a business method of the EJB, the container automatically detects the problem and treats it as a definition error.

Interceptors and decorators may not declare observer methods. If an interceptor or decorator has a method with a parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error.

In addition to the event parameter, observer methods may declare additional parameters, which may declare qualifiers. These additional parameters are injection points.

```
public void afterLogin(@Observes LoggedInEvent event, @Manager User user, Logger log) { ... }
```

10.4.3. Conditional observer methods

A *conditional observer method* is an observer method which is notified of an event only if an instance of the bean that defines the observer method already exists in the current context.

A conditional observer method may be declared by specifying `receive=IF_EXISTS`.

```
public void refreshOnDocumentUpdate(@Observes(receive=IF_EXISTS) @Updated Document doc) { ... }
```

Beans with scope `@Dependent` may not have conditional observer methods. If a bean with scope `@Dependent` has an observer method declared `receive=IF_EXISTS`, the container automatically detects the problem and treats it as a definition error.

The enumeration `javax.enterprise.event.Reception` identifies the possible values of `receive`:

```
public enum Reception { IF_EXISTS, ALWAYS }
```

10.4.4. Transactional observer methods

Transactional observer methods are observer methods which receive event notifications during the before or after completion phase of the transaction in which the event was fired. If no transaction is in progress when the event is fired, they are notified at the same time as other observers.

- A *before completion* observer method is called during the before completion phase of the transaction.
- An *after completion* observer method is called during the after completion phase of the transaction.
- An *after success* observer method is called during the after completion phase of the transaction, only when the transaction completes successfully.
- An *after failure* observer method is called during the after completion phase of the transaction, only when the transaction fails.

The enumeration `javax.enterprise.event.TransactionPhase` identifies the kind of transactional observer method:

```
public enum TransactionPhase {
    IN_PROGRESS,
    BEFORE_COMPLETION,
    AFTER_COMPLETION,
    AFTER_FAILURE,
    AFTER_SUCCESS
}
```

A transactional observer method may be declared by specifying any value other than `IN_PROGRESS` for `during`:

```
void onDocumentUpdate(@Observes(during=AFTER_SUCCESS) @Updated Document doc) { ... }
```

10.5. Observer notification

When an event is fired by the application, the container must:

- determine the observer methods for that event according to the rules of observer resolution defined by Section 10.2, “Observer resolution”, then,
- for each observer method, either invoke the observer method immediately, or register the observer method for later invocation during the transaction completion phase, using a JTA `Synchronization`.

The container calls observer methods as defined in Section 5.5.6, “Invocation of observer methods”.

- If the observer method is a transactional observer method and there is currently a JTA transaction in progress, the container calls the observer method during the appropriate transaction completion phase.
- If the observer method is a conditional observer method and there is no context active for the scope to which the bean declaring the observer method belongs, then the observer method should not be called.
- Otherwise, the container calls the observer immediately.

The order in which observer methods are called is not defined, and so portable applications should not rely upon the order in which observers are called.

Any observer method called before completion of a transaction may call `setRollbackOnly()` to force a transaction rollback. An observer method may not directly initiate, commit or rollback JTA transactions.

Observer methods may throw exceptions:

- If the observer method is a transactional observer method, any exception is caught and logged by the container.
- Otherwise, the exception aborts processing of the event. No other observer methods of that event will be called. The `BeanManager.fireEvent()` or `Event.fire()` method rethrows the exception. If the exception is a checked exception, it is wrapped and rethrown as an (unchecked) `ObserverException`.

For a custom implementation of the `ObserverMethod` interface defined in Section 11.1.3, “The ObserverMethod interface”, the container must call `getTransactionPhase()` to determine if the observer method is transactional observer method, and `notify()` to invoke the method.

Note

CDI 1.1 implementations should call the `notify` method which takes both the event object and the event qualifiers only. The `notify` method which takes only the event object is retained only for backwards compatibility.

10.5.1. Observer method invocation context

The transaction context, client security context and lifecycle contexts active when an observer method is invoked depend upon what kind of observer method it is.

- If the observer method is a before completion transactional observer method, it is called within the context of the transaction that is about to complete and with the same client security context and lifecycle contexts.
- Otherwise, if the observer method is any other kind of transactional observer method, it is called in an unspecified transaction context, but with the same client security context and lifecycle contexts as the transaction that just completed.
- Otherwise, the observer method is called in the same transaction context, client security context and lifecycle contexts as the invocation of `Event.fire()` or `BeanManager.fireEvent()`.

Of course, the transaction and security contexts for a business method of a session bean also depend upon the transaction attribute and `@RunAs` descriptor, if any.

Chapter 11. Portable extensions

A portable extension may integrate with the container by:

- Providing its own beans, interceptors and decorators to the container
- Injecting dependencies into its own objects using the dependency injection service
- Providing a context implementation for a custom scope
- Augmenting or overriding the annotation-based metadata with metadata from some other source

11.1. The `Bean` interface

The `BeanAttributes` interface exposes the basic attributes of a bean.

```
public interface BeanAttributes<T> {
    public Set<Type> getTypes();
    public Set<Annotation> getQualifiers();
    public Class<? extends Annotation> getScope();
    public String getName();
    public Set<Class<? extends Annotation>> getStereotypes();
    public boolean isAlternative();
    public boolean isNullable();
}
```

- `getTypes()`, `getQualifiers()`, `getScope()`, `getName()` and `getStereotypes()` must return the bean types, qualifiers, scope type, bean name and stereotypes of the bean, as defined in Chapter 2, *Concepts*.
- `isAlternative()` must return `true` if the bean is an alternative, and `false` otherwise.
- `isNullable()` must return `true` if the method `Bean.create()` sometimes returns a null value, and `false` otherwise, as defined in Section 5.2.5, “Primitive types and null values”.

The interface `javax.enterprise.inject.spi.Bean` defines everything the container needs to manage instances of a certain bean.

```
public interface Bean<T> extends Contextual<T>, BeanAttributes<T> {
    public Class<?> getBeanClass();
    public Set<InjectionPoint> getInjectionPoints();
}
```

- `getBeanClass()` returns the bean class of the managed bean or session bean or of the bean that declares the producer method or field.
- `getInjectionPoints()` returns a set of `InjectionPoint` objects, defined in Section 5.5.7, “Injection point metadata”, representing injection points of the bean, that will be validated by the container at initialization time.

Note that implementations of `Bean` must also implement the inherited operations defined by the `Contextual` interface defined in Section 6.1, “The Contextual interface”.

An instance of `Bean` exists for every enabled bean.

A portable extension may add support for new kinds of beans beyond those defined by the this specification (managed beans, session beans, producer methods, producer fields and resources) by implementing `Bean` and registering beans with the container, using the mechanism defined in Section 11.5.2, “AfterBeanDiscovery event”.

11.1.1. The `Decorator` interface

The `Bean` object for a decorator must implement the interface `javax.enterprise.inject.spi.Decorator`.

```
public interface Decorator<T> extends Bean<T> {
    public Set<Type> getDecoratedTypes();
    public Type getDelegateType();
    public Set<Annotation> getDelegateQualifiers();
```

```
}
```

- `getDecoratedTypes()` returns the decorated types of the decorator.
- `getDelegateType()` and `getDelegateQualifiers()` return the delegate type and qualifiers of the decorator.

An instance of `Decorator` exists for every enabled decorator.

11.1.2. The `Interceptor` interface

The `Bean` object for an interceptor must implement `javax.enterprise.inject.spi.Interceptor`.

```
public interface Interceptor<T> extends Bean<T> {
    public Set<Annotation> getInterceptorBindings();
    public boolean intercepts(InterceptionType type);
    public Object intercept(InterceptionType type, T instance, InvocationContext ctx) throws Exception;
}
```

- `getInterceptorBindings()` returns the interceptor bindings of the interceptor.
- `intercepts()` returns `true` if the interceptor intercepts the specified kind of lifecycle callback or method invocation, and `false` otherwise.
- `intercept()` invokes the specified kind of lifecycle callback or method invocation interception upon the given instance of the interceptor.

An `InterceptionType` identifies the kind of lifecycle callback, EJB timeout method or business method.

```
public enum InterceptionType {
    AROUND_INVOKE, POST_CONSTRUCT, PRE_DESTROY, PRE_PASSIVATE, POST_ACTIVATE, AROUND_TIMEOUT
}
```

An instance of `Interceptor` exists for every enabled interceptor.

11.1.3. The `ObserverMethod` interface

The interface `javax.enterprise.inject.spi.ObserverMethod` defines everything the container needs to know about an observer method.

```
public interface ObserverMethod<T> {
    public Class<?> getBeanClass();
    public Type getObservedType();
    public Set<Annotation> getObservedQualifiers();
    public Reception getReception();
    public TransactionPhase getTransactionPhase();
    public void notify(T event);
    public void notify(T event, Set<Annotation> qualifiers);
}
```

- `getBeanClass()` returns the class of the type that declares the observer method.
- `getObservedType()` and `getObservedQualifiers()` return the observed event type and qualifiers.
- `getReception()` returns `IF_EXISTS` for a conditional observer and `ALWAYS` otherwise.
- `getTransactionPhase()` returns the appropriate transaction phase for a transactional observer method or `IN_PROGRESS` otherwise.
- `notify()` calls the observer method, as defined in Section 10.5, “Observer notification”.

An instance of `ObserverMethod` exists for every observer method of every enabled bean.

11.2. The `Producer` and `InjectionTarget` interfaces

The interface `javax.enterprise.inject.spi.Producer` provides a generic operation for producing an instance of a type.

```
public interface Producer<T> {
    public T produce(CreationalContext<T> ctx);
    public void dispose(T instance);
    public Set<InjectionPoint> getInjectionPoints();
}
```

For a `Producer` that represents a class:

- `produce()` calls the constructor annotated `@Inject` if it exists, or the constructor with no parameters otherwise, as defined in Section 5.5.1, “Injection using the bean constructor”, and returns the resulting instance. If the class has interceptors, `produce()` is responsible for building the interceptors and decorators of the instance. The instance returned by produce may be a proxy.
- `dispose()` does nothing.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all injected fields, bean constructor parameters and initializer method parameters.

For a `Producer` that represents a producer method or field:

- `produce()` calls the producer method on, or accesses the producer field of, a contextual instance of the bean that declares the producer method, as defined in Section 5.5.4, “Invocation of producer or disposer methods”.
- `dispose()` calls the disposer method, if any, on a contextual instance of the bean that declares the disposer method, as defined in Section 5.5.4, “Invocation of producer or disposer methods”, or performs any additional required cleanup, if any, to destroy state associated with a resource.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all parameters of the producer method.

The subinterface `javax.enterprise.inject.spi.InjectionTarget` provides operations for performing dependency injection and lifecycle callbacks on an instance of a type.

```
public interface InjectionTarget<T> {
        extends Producer<T>
    public void inject(T instance, CreationalContext<T> ctx);
    public void postConstruct(T instance);
    public void preDestroy(T instance);
}
```

- `inject()` performs dependency injection upon the given object. The container performs Java EE component environment injection, according to the semantics required by the Java EE platform specification, sets the value of all injected fields, and calls all initializer methods, as defined in Section 5.5.2, “Injection of fields and initializer methods”.
- `postConstruct()` calls the `@PostConstruct` callback, if it exists, according to the semantics required by the Java EE platform specification.
- `preDestroy()` calls the `@PreDestroy` callback, if it exists, according to the semantics required by the Java EE platform specification.

Implementations of `Producer` and `InjectionTarget` must ensure that the set of injection points returned by `getInjectionPoints()` are injected by `produce()` or `inject()`.

11.3. The `BeanManager` object

The interface `javax.enterprise.inject.spi.BeanManager` provides operations for obtaining contextual references for beans, along with many other operations of use to portable extensions.

The container provides a built-in bean with bean type `BeanManager`, scope `@Dependent` and qualifier `@Default`. The built-in implementation must be a passivation capable dependency, as defined in Section 6.6.2, “Passivation capable dependencies”. Thus, any bean may obtain an instance of `BeanManager` by injecting it:

```
@Inject BeanManager manager;
```

Any operation of `BeanManager` may be called at any time during the execution of the application.

11.3.1. Obtaining a reference to the CDI container

Portable extensions and other objects sometimes interact directly with the container via programmatic API call. The abstract `javax.enterprise.inject.spi.CDI` provides access to the `BeanManager` as well providing lookup of bean instances.

```
public abstract class CDI<T> implements Instance<T> {
   public static CDI<Object> current() { ... }
   public static void setCDIProvider(CDIProvider provider);
   public abstract BeanManager getBeanManager();
}
```

A portable extension or other object may obtain a reference to the current container by calling `CDI.current()`. `CDI.getBeanManager()` may be called at any time after the container fires the `BeforeBeanDiscovery` container lifecycle event until the container fires the `BeforeShutdown` container lifecycle event. Other methods on `CDI` may be called after the application initialization is completed until the application shutdown starts. If methods on `CDI` are called at any other time, non-portable behavior results.

When `CDI.current()` is called, `getCDI()` method is called on `javax.enterprise.inject.spi.CDIProvider`.

The `CDIProvider` to use may be set by the application or container using the `setCDIProvider()` method. If the `setCDIProvider()` has not been called, the first service provider of the service `javax.enterprise.inject.spi.CDIProvider` declared in META-INF/services is used. If no provider is available an IllegalStateException is thrown.

```
public interface CDIProvider {
   public CDI<Object> getCDI();
}
```

A Java EE container is required to provide a CDI provider that will allow access to the current container for any Java EE application or Java EE module which contains enabled beans.

Java EE components may obtain an instance of `BeanManager` from JNDI by looking up the name `java:comp/BeanManager`.

11.3.2. Obtaining a contextual reference for a bean

The method `BeanManager.getReference()` returns a contextual reference for a given bean and bean type, as defined in Section 6.5.3, “Contextual reference for a bean”.

```
public Object getReference(Bean<?> bean, Type beanType, CreationalContext<?> ctx);
```

The first parameter is the `Bean` object representing the bean. The second parameter represents a bean type that must be implemented by any client proxy that is returned. The third parameter is an instance of `CreationalContext` that may be used to destroy any object with scope `@Dependent` that is created.

If the given type is not a bean type of the given bean, an `IllegalArgumentException` is thrown.

11.3.3. Obtaining an injectable reference

The method `BeanManager.getInjectableReference()` returns an injectable reference for a given injection point, as defined in Section 6.5.5, “Injectable references”.

```
public Object getInjectableReference(InjectionPoint ij, CreationalContext<?> ctx);
```

The first parameter represents the target injection point. The second parameter is an instance of `CreationalContext` that may be used to destroy any object with scope `@Dependent` that is created.

If the `InjectionPoint` represents a decorator delegate injection point, `getInjectableReference()` returns a delegate, as defined in Section 8.1.2, “Decorator delegate injection points”.

If typesafe resolution results in an unsatisfied dependency, the container must throw an `UnsatisfiedResolutionException`. If typesafe resolution results in an unresolvable ambiguous dependency, the container must throw an `AmbiguousRes-`

olutionException.

Implementations of Bean usually maintain a reference to an instance of BeanManager. When the Bean implementation performs dependency injection, it must obtain the contextual instances to inject by calling BeanManager.getInjectableReference(), passing an instance of InjectionPoint that represents the injection point and the instance of CreationalContext that was passed to Bean.create().

11.3.4. Obtaining a CreationalContext

An instance of CreationalContext for a certain instance of Contextual may be obtained by calling BeanManager.createCreationalContext().

```
public <T> CreationalContext<T> createCreationalContext(Contextual<T> contextual);
```

An instance of CreationalContext for a non-contextual object may be obtained by passing a null value to createCreationalContext().

11.3.5. Obtaining a Bean by type

The method BeanManager.getBeans() returns the set of beans which have the given required type and qualifiers and are available for injection in the module or library containing the class into which the BeanManager was injected or the Java EE component from whose JNDI environment namespace the BeanManager was obtained, according to the rules for candidates of typesafe resolution defined in Section 5.2.1, “Performing typesafe resolution”.

```
public Set<Bean<?>> getBeans(Type beanType, Annotation... qualifiers);
```

The first parameter is a required bean type. The remaining parameters are required qualifiers.

If no qualifiers are passed to getBeans(), the default qualifier @Default is assumed.

If the given type represents a type variable, an IllegalArgumentException is thrown.

If two instances of the same qualifier type are given, an IllegalArgumentException is thrown.

If an instance of an annotation that is not a qualifier type is given, an IllegalArgumentException is thrown.

11.3.6. Obtaining a Bean by name

The method BeanManager.getBeans() which accepts a string returns the set of beans which have the given bean name and are available for injection in the module or library containing the class into which the BeanManager was injected or the Java EE component from whose JNDI environment namespace the BeanManager was obtained, according to the rules of name resolution defined in Section 5.3, “EL name resolution”.

```
public Set<Bean<?>> getBeans(String name);
```

The parameter is a bean name.

11.3.7. Obtaining a passivation capable bean by identifier

The method BeanManager.getPassivationCapableBean() returns the PassivationCapable bean with the given identifier (see Section 6.6.1, “Passivation capable beans”).

```
public Bean<?> getPassivationCapableBean(String id);
```

11.3.8. Resolving an ambiguous dependency

The method BeanManager.resolve() applies the ambiguous dependency resolution rules defined in Section 5.2.2, “Unsatisfied and ambiguous dependencies” to a set of Beans.

```
public <X> Bean<? extends X> resolve(Set<Bean<? extends X>> beans);
```

If the ambiguous dependency resolution rules fail (as defined in Section 5.2.2, “Unsatisfied and ambiguous dependencies”, the container must throw an `AmbiguousResolutionException`.

`BeanManager.resolve()` must return null if:

- null is passed to `resolve()`, or
- no beans are passed to `resolve()`, or
- no bean is available for injection in the module (as defined in Section 5.1, “Modularity”)

11.3.9. Validating an injection point

The `BeanManager.validate()` operation validates an injection point and throws an `InjectionException` if there is a deployment problem (for example, an unsatisfied or unresolvable ambiguous dependency) associated with the injection point.

```
public void validate(InjectionPoint injectionPoint);
```

11.3.10. Firing an event

The method `BeanManager.fireEvent()` fires an event and notifies observers, according to Section 10.5, “Observer notification”.

```
public void fireEvent(Object event, Annotation... qualifiers);
```

The first argument is the event object. The remaining parameters are event qualifiers.

If the runtime type of the event object contains a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same qualifier type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a qualifier type is given, an `IllegalArgumentException` is thrown.

If the runtime type of the event object is assignable to the type of a container lifecycle event, `IllegalArgumentException` is thrown.

11.3.11. Observer method resolution

The method `BeanManager.resolveObserverMethods()` resolves observer methods for an event according to the rules of observer resolution defined in Section 10.2, “Observer resolution”.

```
public <T> Set<ObserverMethod<? super T>> resolveObserverMethods(T event, Annotation... qualifiers);
```

The first parameter of `resolveObserverMethods()` is the event object. The remaining parameters are event qualifiers.

If the runtime type of the event object contains a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same qualifier type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a qualifier type is given, an `IllegalArgumentException` is thrown.

11.3.12. Decorator resolution

The method `BeanManager.resolveDecorators()` returns the ordered list of decorators for a set of bean types and a set of qualifiers and which are enabled in the module or library containing the class into which the `BeanManager` was injected or the Java EE component from whose JNDI environment namespace the `BeanManager` was obtained, as defined in Section 8.3, “Decorator resolution”.

```
List<Decorator<?>> resolveDecorators(Set<Type> types, Annotation... qualifiers);
```

The first argument is the set of bean types of the decorated bean. The annotations are qualifiers declared by the decorated bean.

If two instances of the same qualifier type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a qualifier type is given, an `IllegalArgumentException` is thrown.

If the set of bean types is empty, an `IllegalArgumentException` is thrown.

11.3.13. Interceptor resolution

The method `BeanManager.resolveInterceptors()` returns the ordered list of interceptors for a set of interceptor bindings and a type of interception and which are enabled in the module or library containing the class into which the `BeanManager` was injected or the Java EE component from whose JNDI environment namespace the `BeanManager` was obtained, as defined in Section 9.5, “Interceptor resolution”.

```
List<Interceptor<?>> resolveInterceptors(InterceptionType type,
                                         Annotation... interceptorBindings);
```

If two instances of the same interceptor binding type are given, an `IllegalArgumentException` is thrown.

If no interceptor binding type instance is given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not an interceptor binding type is given, an `IllegalArgumentException` is thrown.

11.3.14. Determining if an annotation is a qualifier type, scope type, stereotype or interceptor binding type

A portable extension may test an annotation to determine if it is a qualifier type, scope type, stereotype or interceptor binding type, obtain the set of meta-annotations declared by a stereotype or interceptor binding type, or determine if a scope type is a normal or passivating scope.

```
public boolean isScope(Class<? extends Annotation> annotationType);
public boolean isQualifier(Class<? extends Annotation> annotationType);
public boolean isInterceptorBinding(Class<? extends Annotation> annotationType);
public boolean isStereotype(Class<? extends Annotation> annotationType);

public boolean isNormalScope(Class<? extends Annotation> scopeType);
public boolean isPassivatingScope(Class<? extends Annotation> scopeType);
public Set<Annotation> getInterceptorBindingDefinition(Class<? extends Annotation> qualifierType);
public Set<Annotation> getStereotypeDefinition(Class<? extends Annotation> stereotype);
```

11.3.15. Determining the hash code and equivalence of qualifiers and interceptor bindings

A portable extension may determine if two qualifiers or two interceptor bindings are considered equivalent for the purposes of typesafe resolution, as defined in Section 5.2.1, “Performing typesafe resolution”.

```
public boolean areQualifiersEquivalent(Annotation qualifier1, Annotation qualifier2);
public boolean areInterceptorBindingsEquivalent(Annotation interceptorBinding1, Annotation interceptorBinding2);
```

A portable extension may determine the hash code of a qualifier or and interceptor binding, ignoring any members annotated with `@Nonbinding`.

```
public int getQualifierHashCode(Annotation qualifier);
public int getInterceptorBindingHashCode(Annotation interceptorBinding);
```

11.3.16. Obtaining the active `Context` for a scope

The method `BeanManager.getContext()` retrieves an active context object associated with the a given scope, as defined in Section 6.5.1, “The active context object for a scope”.

```
public Context getContext(Class<? extends Annotation> scopeType);
```

11.3.17. Obtaining the `ELResolver`

The method `BeanManager.getELResolver()` returns the `javax.el.ELResolver` specified in Section 12.5, “Integration with Unified EL”.

```
public ELResolver getELResolver();
```

11.3.18. Wrapping a Unified EL `ExpressionFactory`

The method `BeanManager.wrapExpressionFactory()` returns a wrapper `javax.el.ExpressionFactory` that delegates `MethodExpression` and `ValueExpression` creation to the given `ExpressionFactory`. When a Unified EL expression is evaluated using a `MethodExpression` or `ValueExpression` returned by the wrapper `ExpressionFactory`, the rules defined in Section 6.4.3, “Dependent pseudo-scope and Unified EL” are enforced by the container.

```
public ExpressionFactory wrapExpressionFactory(ExpressionFactory expressionFactory);
```

11.3.19. Obtaining an `AnnotatedType` for a class

The method `BeanManager.createAnnotatedType()` returns an `AnnotatedType` that may be used to read the annotations of the given Java class or interface.

The methods `BeanManager.getAnnotatedType()` and `BeanManager.getAnnotatedTypes()` returns the `AnnotatedTypes` discovered or added during container initialization.

The method `BeanManager.getAnnotatedTypes()` returns the `AnnotatedTypes` discovered by or added to the container during container initialization.

```
public <T> AnnotatedType<T> createAnnotatedType(Class<T> type);
public <T> AnnotatedType<T> getAnnotatedType(Class<T> type, String id);
public <T> Iterable<AnnotatedType<T>> getAnnotatedTypes(Class<T> type);
```

11.3.20. Obtaining an `InjectionTarget` for a class

The method `BeanManager.createInjectionTarget()` returns a container provided implementation of `InjectionTarget` for a given `AnnotatedType` or throws an `IllegalArgumentException` if there is a definition error associated with any injection point of the type.

```
public <T> InjectionTarget<T> createInjectionTarget(AnnotatedType<T> type);
```

11.3.21. Obtaining a `Producer` for a field or method

The method `BeanManager.createProducer()` returns a container provided implementation of `Producer` for a given `AnnotatedMethod` or `AnnotatedField` or throws an `IllegalArgumentException` if there is a definition error associated with the producer method or field.

```
public <X> Producer<?> createProducer(AnnotatedField<? super X> field, Bean<X> declaringBean);
public <X> Producer<?> createProducer(AnnotatedMethod<? super X> method, Bean<X> declaringBean);
```

11.3.22. Obtaining an `InjectionPoint`

The method `BeanManager.createInjectionPoint()` returns a container provided implementation of `InjectionPoint` for a given `AnnotatedField` or `AnnotatedParameter` or throws an `IllegalArgumentException` if there is a definition error associated with the injection point.

```
public InjectionPoint createInjectionPoint(AnnotatedField<?> field);
public InjectionPoint createInjectionPoint(AnnotatedParameter<?> parameter);
```

11.3.23. Obtaining a `BeanAttributes`

The method `BeanManager.createBeanAttributes()` returns a container provided implementation of `BeanAttributes` by reading the annotations of a given `AnnotatedType` or `AnnotatedMember`, according to the rules define in Chapter 2, *Concepts*, or throws an `IllegalArgumentException` if there is a definition error associated with the declared bean attributes.

```
public <T> BeanAttributes<T> createBeanAttributes(AnnotatedType<T> type);
public BeanAttributes<?> createBeanAttributes(AnnotatedMember<?> member);
```

11.3.24. Obtaining a `Bean`

The method `BeanManager.createBean()` returns a container provided implementation of `Bean`. The method accepts:

- a `BeanAttributes`, which determines the bean types, qualifiers, scope, name and stereotypes of the returned `Bean`, and the return values of `isAlternative()` and `isNullable()`, and
- a class, which determines the return value of `Bean.getClass()`.

The first version of the method also accepts:

- an `InjectionTarget`, which is used to create and destroy instances of the bean, to perform dependency injection and lifecycle callbacks, and which determines the return value of `Bean.getInjectionPoints()`.

```
public <T> Bean<T> createBean(BeanAttributes<T> attributes, Class<T> beanClass,
                          InjectionTarget<T> injectionTarget);
```

The second version of the method also accepts:

- a `Producer`, which is used to create and destroy instances of the bean, and which determines the return value of `Bean.getInjectionPoints()`.

```
public <T> Bean<T> createBean(BeanAttributes<T> attributes, Class<?> beanClass,
                          Producer<T> producer);
```

11.3.25. Obtaining the instance of an `Extension`

The method `BeanManager.getExtensions()` returns the container's instance of an `Extension` class declared in `META-INF/services`, or throws an `IllegalArgumentException` if the container has no instance of the given class.

```
public <T extends Extension> T getExtension(Class<T> extensionClass);
```

11.4. Alternative metadata sources

A portable extension may provide an alternative metadata source, such as configuration by XML.

The interfaces `AnnotatedType`, `IdentifiedAnnotatedType`, `AnnotatedField`, `AnnotatedMethod`, `AnnotatedConstructor` and `AnnotatedParameter` in the package `javax.enterprise.inject.spi` allow a portable extension to specify metadata that overrides the annotations that exist on a bean class. The portable extension is responsible for implementing the interfaces, thereby exposing the metadata to the container.

In general, the behavior is as defined by the Java Language Specification, and only deviations from the Java Language Specification are noted.

The interface `javax.enterprise.inject.spi.AnnotatedType` exposes the `Class` object and members.

```
public interface AnnotatedType<X>
        extends Annotated {
    public Class<X> getJavaClass();
    public Set<AnnotatedConstructor<X>> getConstructors();
    public Set<AnnotatedMethod<? super X>> getMethods();
    public Set<AnnotatedField<? super X>> getFields();
}
```

- `getConstructors()` returns all default-access, public, protected or private constructors declared for the type.

- `getMethods()` returns all default-access, public, protected or private methods declared on the type and those declared on any supertypes. The container should call `AnnotatedMethod.getJavaMember().getDeclaringClass()` to determine the type in the type hierarchy that declared the method.
- `getFields()` returns all default-access, public, protected or private fields declared on the type and those declared on any supertypes. The container should call `AnnotatedField.getJavaMember().getDeclaringClass()` to determine the type in the type hierarchy that declared the field.

When determining annotations on a type, the container must only consider the special inheritance rules defined for scope types in Section 4.1, “Inheritance of type-level metadata”.

The interface `javax.enterprise.inject.spi.IdentifiedAnnotatedType` allows multiple annotated types, based on the same underlying type, to be defined.

```
public interface IdentifiedAnnotatedType<X>
        extends AnnotatedType<X> {
    public String getId();
}
```

- `getId()` returns the identifier for the type. The identifier returned should be unique.

If an `AnnotatedType` is an instance of `IdentifiedAnnotatedType` then `IdentifiedAnnotatedType.getId()` is used to identify the annotated type to the container, otherwise the fully qualified class name of `AnnotatedType.getJavaClass()` is used to identify the type. `AnnotatedTypes` discovered by the container use the fully qualified class name of `AnnotatedType.getJavaClass()` to identify the type.

The interface `javax.enterprise.inject.spi.AnnotatedField` exposes the `Field` object.

```
public interface AnnotatedField<X>
        extends AnnotatedMember<X> {
    public Field getJavaMember();
}
```

The interface `javax.enterprise.inject.spi.AnnotatedMethod` exposes the `Method` object.

```
public interface AnnotatedMethod<X>
        extends AnnotatedCallable<X> {
    public Method getJavaMember();
}
```

The interface `javax.enterprise.inject.spi.AnnotatedConstructor` exposes the `Constuctor` object.

```
public interface AnnotatedConstructor<X>
        extends AnnotatedCallable<X> {
    public Constructor<X> getJavaMember();
}
```

The interface `javax.enterprise.inject.spi.AnnotatedParameter` exposes the `position` of the parameter object and the declaring program element.

```
public interface AnnotatedParameter<X>
        extends Annotated {
    public int getPosition();
    public AnnotatedCallable<X> getDeclaringCallable();
}
```

The interface `javax.enterprise.inject.spi.AnnotatedMemember` exposes the `Member` object and the `AnnotatedType` that defines the member.

```
public interface AnnotatedMember<X>
        extends Annotated {
    public Member getJavaMember();
    public boolean isStatic();
    public AnnotatedType<X> getDeclaringType();
}
```

The interface `javax.enterprise.inject.spi.AnnotatedCallable` exposes the parameters of an invokable object.

Contexts and Dependency Injection for Java EE 1.1 deprecated the method `AnnotatedMember.isStatic`. The container should instead call `AnnotatedMember.getJavaMember().getModifiers()` to determine if the member is static.

```
public interface AnnotatedCallable<X>
        extends AnnotatedMember<X> {
    public List<AnnotatedParameter<X>> getParameters();
}
```

The interface `javax.enterprise.inject.spi.Annotated` exposes the overriding annotations and type declarations.

```
public interface Annotated {
    public Type getBaseType();
    public Set<Type> getTypeClosure();
    public <T extends Annotation> T getAnnotation(Class<T> annotationType);
    public Set<Annotation> getAnnotations();
    public boolean isAnnotationPresent(Class<? extends Annotation> annotationType);
}
```

- `getBaseType()` returns the type of the program element.
- `getTypeClosure()` returns all types to which the base type should be considered assignable.
- `getAnnotation()` returns the program element annotation of the given annotation type, or a null value.
- `getAnnotations()` returns all annotations of the program element.
- `isAnnotationPresent()` returns `true` if the program element has an annotation of the given annotation type, or `false` otherwise.

The container must use the operations of `Annotated` and its subinterfaces to discover program element types and annotations. The container must not directly call the Java Reflection API. In particular, the container must:

- call `Annotated.getBaseType()` to determine the type of an injection point, event parameter or disposed parameter,
- call `Annotated.getTypeClosure()` to determine the bean types of any kind of bean,
- call `Annotated.getAnnotations()` to determine the scope, qualifiers, stereotypes and interceptor bindings of a bean,
- call `Annotated.isAnnotationPresent()` and `Annotated.getAnnotation()` to read any bean annotations defined by this specification, and
- call `AnnotatedType.getConstructors()`, `AnnotatedType.getMethods()` and `AnnotatedType.getFields()` to determine the members of a bean class.

11.5. Container lifecycle events

During the application initialization process, the container fires a series of events, allowing portable extensions to integrate with the container initialization process defined in Section 12.2, “Application initialization lifecycle”.

Observer methods of these events must belong to *extensions*. An extension is a service provider of the service `javax.enterprise.inject.spi.Extension` declared in `META-INF/services`.

```
public interface Extension {}
```

Service providers may have observer methods, which may observe any event, including any container lifecycle event, and obtain an injected `BeanManager` reference. Any decorators associated with `BeanManager` will not be applied. If other beans are injected into an extension's observer methods, non-portable behavior results. An extension may use `BeanManager.fireEvent()` to deliver events to observer methods defined on extensions. The container is not required to deliver events fired during application initialization to observer methods defined on beans.

The container instantiates a single instance of each extension at the beginning of the application initialization process and maintains a reference to it until the application shuts down. The container delivers event notifications to this instance by calling its observer methods.

For each service provider, the container must provide a bean of scope `@ApplicationScoped` and qualifier `@Default`, supporting injection of a reference to the service provider instance. The bean types of this bean include the class of the service provider and all superclasses and interfaces.

11.5.1. `BeforeBeanDiscovery` event

The container must fire an event before it begins the bean discovery process. The event object must be of type `javax.enterprise.inject.spi.BeforeBeanDiscovery`:

```
public interface BeforeBeanDiscovery {
    public void addQualifier(Class<? extends Annotation> qualifier);
    public void addScope(Class<? extends Annotation> scopeType, boolean normal, boolean passivating);
    public void addStereotype(Class<? extends Annotation> stereotype, Annotation... stereotypeDef);
    public void addInterceptorBinding(Class<? extends Annotation> bindingType, Annotation... bindingTypeDef);
    public void addAnnotatedType(AnnotatedType<?> type);
}
```

- `addQualifier()` declares an annotation type as a qualifier type.
- `addScope()` declares an annotation type as a scope type.
- `addStereotype()` declares an annotation type as a stereotype, and specifies its meta-annotations.
- `addInterceptorBinding()` declares an annotation type as an interceptor binding type, and specifies its meta-annotations.
- `addAnnotatedType()` adds a given `AnnotatedType` to the set of types which will be scanned during bean discovery. If an `AnnotatedType` with the same identifier already exists in the set of types the container automatically detects the problem and treats it as a deployment problem.

  Portable extensions are encouraged to add `IdentifiedAnnotatedTypes` in order to not accidentally conflict with discovered types.

```
void beforeBeanDiscovery(@Observes BeforeBeanDiscovery event) { ... }
```

If any observer method of the `BeforeBeanDiscovery` event throws an exception, the exception is treated as a definition error by the container.

11.5.2. `AfterBeanDiscovery` event

The container must fire a second event when it has fully completed the bean discovery process, validated that there are no definition errors relating to the discovered beans, and registered `Bean` and `ObserverMethod` objects for the discovered beans, but before detecting deployment problems.

The event object must be of type `javax.enterprise.inject.spi.AfterBeanDiscovery`:

```
public interface AfterBeanDiscovery {
    public void addDefinitionError(Throwable t);
    public void addBean(Bean<?> bean);
    public void addObserverMethod(ObserverMethod<?> observerMethod);
    public void addContext(Context context);
}
```

- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after all observers have been notified.
- `addBean()` fires an event of type `ProcessBean` containing the given `Bean` and then registers the `Bean` with the container, thereby making it available for injection into other beans. The given `Bean` may implement `Interceptor` or `Decorator`.
- `addObserverMethod()` fires an event of type `ProcessObserverMethod` containing the given `ObserverMethod` and then registers the `ObserverMethod` with the container, thereby making it available for event notifications.
- `addContext()` registers a custom `Context` object with the container.

A portable extension may take advantage of this event to register beans, interceptors, decorators, observer methods and custom context objects with the container.

```
void afterBeanDiscovery(@Observes AfterBeanDiscovery event, BeanManager manager) { ... }
```

If any observer method of the `AfterBeanDiscovery` event throws an exception, the exception is treated as a definition error by the container.

11.5.3. `AfterDeploymentValidation` event

The container must fire a third event after it has validated that there are no deployment problems and before creating contexts or processing requests.

The event object must be of type `javax.enterprise.inject.spi.AfterDeploymentValidation`:

```
public interface AfterDeploymentValidation {
    public void addDeploymentProblem(Throwable t);
}
```

- `addDeploymentProblem()` registers a deployment problem with the container, causing the container to abort deployment after all observers have been notified.

```
void afterDeploymentValidation(@Observes AfterDeploymentValidation event, BeanManager manager) { ... }
```

If any observer method of the `AfterDeploymentValidation` event throws an exception, the exception is treated as a deployment problem by the container.

The container must not allow any request to be processed by the deployment until all observers of this event return.

11.5.4. `BeforeShutdown` event

The container must fire a final event after it has finished processing requests and destroyed all contexts.

The event object must be of type `javax.enterprise.inject.spi.BeforeShutdown`:

```
public interface BeforeShutdown {}
```

```
void beforeShutdown(@Observes BeforeShutdown event, BeanManager manager) { ... }
```

If any observer method of the `BeforeShutdown` event throws an exception, the exception is ignored by the container.

11.5.5. `ProcessModule` event

The container must fire an event for each bean archive, before it processes the classes packaged in that module.

The event object must be of type `javax.enterprise.inject.spi.ProcessModule`.

```
public interface ProcessModule {
    public Set<Class<?>> getAlternatives();
    public List<Class<?>> getInterceptors();
    public List<Class<?>> getDecorators();
    public Iterator<AnnotatedType<?>> getAnnotatedTypes();
    public InputStream getBeansXml();
}
```

- `getAlternatives()` returns the set of enabled alternatives of the bean deployment archive.
- `getInterceptors()` returns the list of enabled interceptors of the bean deployment archive.
- `getDecorators()` returns the list of enabled decorators of the bean deployment archive.
- `getAnnotatedTypes()` returns an iterator over `AnnotatedType` objects representing the Java classes and interfaces in the bean archive.
- `getBeansXml()` returns an input stream which can be used to read in the `beans.xml` for this module.

Any observer of this event is permitted to add classes to, or remove classes from, the set of alternatives, list of interceptors or list of decorators. The container must use the final values of these collections, after all observers have been called, to determine the enabled alternatives, interceptors, and decorators for the bean deployment archive. The initial values of these collections is determined by reading the `beans.xml` file of the bean deployment archive.

If any observer method of a `ProcessModule` event throws an exception, the exception is treated as a deployment problem by the container.

11.5.6. `ProcessAnnotatedType` event

The container must fire an event, before it processes a type, for each:

- Java class, interface, enum or annotation discovered in a bean archive,
- Annotated type added by `BeforeBeanDiscovery.addAnnotatedType()`,
- Annotation added by `BeforeBeanDiscovery.addInterceptorBinding()`, `BeforeBeanDiscovery.addQualifier()`, `BeforeBeanDiscovery.addScope()` or `BeforeBeanDiscovery.addStereotype()`.

An event is not fired for any type annotated with `@Vetoed`, or in a package annotated with `@Vetoed`.

The event object must be of type `javax.enterprise.inject.spi.ProcessAnnotatedType<X>`, where `X` is the class, for types discovered in a bean archive, or of type `javax.enterprise.inject.spi.ProcessSyntheticAnnotatedType<X>` for types added by `BeforeBeanDiscovery.addAnnotatedType()`, `BeforeBeanDiscovery.addInterceptorBinding()`, `BeforeBeanDiscovery.addQualifier()`, `BeforeBeanDiscovery.addScope()` or `BeforeBeanDiscovery.addStereotype()`.

The annotation `@WithAnnotations` may be applied to the event parameter. If the annotation is applied, the container must only deliver `ProcessAnnotatedType` events for types which contain at least one of the annotations specified. The annotation can appear on the type, on any field, method or constructor declared by the type, or on any parameter of any method or constructor declared by the type. The annotation may be applied as a meta-annotation on any annotation considered.

If the `@WithAnnotations` annotation is applied to any other event parameter, the container automatically detects the problem and treats it as a definition error.

```
public interface ProcessAnnotatedType<X> {
    public AnnotatedType<X> getAnnotatedType();
    public void setAnnotatedType(AnnotatedType<X> type);
    public void veto();
}
```

```
interface ProcessSyntheticAnnotatedType<X> extends ProcessAnnotatedType<X> {
    public Extension getSource();
}
```

- `getAnnotatedType()` returns the `AnnotatedType` object that will be used by the container to read the declared annotations.
- `setAnnotatedType()` replaces the `AnnotatedType`.
- `veto()` forces the container to ignore the type.
- `getSource()` returns the `Extension` instance that added the annotated type.

Any observer of this event is permitted to wrap and/or replace the `AnnotatedType`. The container must use the final value of this property, after all observers have been called, as the only source of types and annotations for the the program elements.

For example, the following observer decorates the `AnnotatedType` for every class that is discovered by the container.

```
<T> void decorateAnnotatedType(@Observes ProcessAnnotatedType<T> pat) {
    pat.setAnnotatedType( decorate( pat.getAnnotatedType() ) );
}
```

If any observer method of a `ProcessAnnotatedType` event throws an exception, the exception is treated as a definition error by the container.

11.5.7. `ProcessInjectionPoint` event

The container must fire an event for every injection point of every Java EE component class supporting injection that may be instantiated by the container at runtime, including every managed bean declared using `@ManagedBean`, EJB session or message-driven bean, enabled bean, enabled interceptor or enabled decorator.

The event object must be of type `javax.enterprise.inject.spi.ProcessInjectionPoint<T, X>` where `T` is the managed bean class, session bean class or Java EE component class supporting injection, and `X` is the declared type of the injection point.

```
public interface ProcessInjectionPoint<T, X> {
    public InjectionPoint getInjectionPoint();
    public void setInjectionPoint(InjectionPoint injectionPoint);
    public void addDefinitionError(Throwable t);
}
```

- `getInjectionPoint()` returns the `InjectionPoint` object that will be used by the container to perform injection.
- `setInjectionPoint()` replaces the `InjectionPoint`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `InjectionPoint`. The container must use the final value of this property, after all observers have been called, whenever it performs injection upon the injection point.

If any observer method of a `ProcessInjectionPoint` event throws an exception, the exception is treated as a definition error by the container.

11.5.8. `ProcessInjectionTarget` event

The container must fire an event for every enum and every Java EE component class supporting injection that may be instantiated by the container at runtime, including every managed bean declared using `@ManagedBean`, EJB session or message-driven bean, enabled bean, enabled interceptor or enabled decorator.

The event object must be of type `javax.enterprise.inject.spi.ProcessInjectionTarget<X>`, where `X` is the managed bean class, session bean class or Java EE component class supporting injection.

```
public interface ProcessInjectionTarget<X> {
    public AnnotatedType<X> getAnnotatedType();
    public InjectionTarget<X> getInjectionTarget();
    public void setInjectionTarget(InjectionTarget<X> injectionTarget);
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotatedType()` returns the `AnnotatedType` representing the managed bean class, session bean class or other Java EE component class supporting injection.
- `getInjectionTarget()` returns the `InjectionTarget` object that will be used by the container to perform injection.
- `setInjectionTarget()` replaces the `InjectionTarget`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `InjectionTarget`. The container must use the final value of this property, after all observers have been called, whenever it performs injection upon the managed bean, session bean or other Java EE component class supporting injection.

For example, this observer decorates the `InjectionTarget` for all servlets.

```
<T extends Servlet> void decorateServlet(@Observes ProcessInjectionTarget<T> pit) {
    pit.setInjectionTarget( decorate( pit.getInjectionTarget() ) );
}
```

If any observer method of a `ProcessInjectionTarget` event throws an exception, the exception is treated as a definition error by the container.

11.5.9. `ProcessProducer` event

The container must fire an event for each producer method or field of each enabled bean, including resources.

The event object must be of type `javax.enterprise.inject.spi.ProcessProducer<T, X>`, where `T` is the bean class of the bean that declares the producer method or field and `X` is the return type of the producer method or the type of the producer field.

```
public interface ProcessProducer<T, X> {
    public AnnotatedMember<T> getAnnotatedMember();
    public Producer<X> getProducer();
    public void setProducer(Producer<X> producer);
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotatedMember()` returns the `AnnotatedField` representing the producer field or the `AnnotatedMethod` representing the producer method.
- `getProducer()` returns the `Producer` object that will be used by the container to call the producer method or read the producer field.
- `setProducer()` replaces the `Producer`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `Producer`. The container must use the final value of this property, after all observers have been called, whenever it calls the producer or disposer.

For example, this observer decorates the `Producer` for all producer methods and fields of type `EntityManager`.

```
void decorateEntityManager(@Observes ProcessProducer<?, EntityManager> pp) {
    pit.setProducer( decorate( pp.getProducer() ) );
}
```

If any observer method of a `ProcessProducer` event throws an exception, the exception is treated as a definition error by the container.

For producer methods, the event must be of type `ProcessProducerMethod`.

```
public interface ProcessProducerMethod<T, X> extends ProcessProducer<T, X> {
    public AnnotatedMethod<T> getAnnotatedMember();
}
```

For producer fields, the event must be of type `ProcessProducerField`.

```
public interface ProcessProducerField<T, X> extends ProcessProducer<T, X> {
    public AnnotatedField<T> getAnnotatedMember();
    public void setInitializer(Initializer<X> producer);
}
```

- `setInitializer()` sets the `Initializer` object that will be used by the container to obtain the initial value of the producer field.

The interface `Initializer` lets a portable extension provide an initial value for a producer field.

```
public interface Initilizer<X> {
    public X getInitialValue(AnnotatedField<?> field);
}
```

- `getInitialValue()` returns the initial value of the producer field. If the producer field is non-static, the container must inject this value to the producer field when it performs Java EE component environment injection upon an instance of the bean that declares the producer field. The container must use this value as the value of the producer field, whether static or non-static, unless the application explicitly assigns a value to the field before the field is accessed by the container.

11.5.10. `ProcessBeanAttributes` event

The container must fire an event for each enabled bean, interceptor or decorator deployed in a bean archive, before registering the `Bean` object. No event is fired for any `@New` qualified bean, defined in Section 3.14, “@New qualified beans”.

The event object must be of type `javax.enterprise.inject.spi.ProcessBeanAttributes<T>` where `T` is the bean class of the managed bean or session bean, the return type of the producer method, or the type of the producer field.

Resources are considered to be producer fields.

```
public interface ProcessBeanAttributes<T> {
    public Annotated getAnnotated();
    public BeanAttributes<T> getBeanAttributes();
    public void setBeanAttributes(BeanAttributes<T> beanAttributes);
    public void addDefinitionError(Throwable t);
    public void veto();
}
```

- `getAnnotated()` returns the `AnnotatedType` representing the managed bean class or session bean class, the `AnnotatedMethod` representing the producer field, or the `AnnotatedField` representing the producer field.
- `getBeanAttributes()` returns the `BeanAttributes` object that will be used by the container to manage instances of the bean.
- `setBeanAttributes()` replaces the `BeanAttributes`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.
- `veto()` forces the container to ignore the bean.

Any observer of this event is permitted to wrap and/or replace the `BeanAttributes`. The container must use the final value of this property, after all observers have been called, to manage instances of the bean.

If any observer method of a `ProcessBeanAttributes` event throws an exception, the exception is treated as a definition error by the container.

11.5.11. `ProcessBean` event

The container must fire an event for each enabled bean, interceptor or decorator deployed in a bean archive, after firing the `ProcessBeanAttributes` for the bean and before registering the `Bean` object. No event is fired for any `@New` qualified bean, defined in Section 3.14, “@New qualified beans”.

The event object type in the package `javax.enterprise.inject.spi` depends upon what kind of bean was discovered:

- For a managed bean with bean class `X`, the container must raise an event of type `ProcessManagedBean<X>`.
- For a session bean with bean class `X`, the container must raise an event of type `ProcessSessionBean<X>`.
- For a producer method with method return type `X` of a bean with bean class `T`, the container must raise an event of type `ProcessProducerMethod<T, X>`.
- For a producer field with field type `X` of a bean with bean class `T`, the container must raise an event of type `ProcessProducerField<T, X>`.

Resources are considered to be producer fields.

The interface `javax.enterprise.inject.spi.ProcessBean` is a supertype of all these event types:

```
public interface ProcessBean<X> {
    public Annotated getAnnotated();
```

```
    public Bean<X> getBean();
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotated()` returns the `AnnotatedType` representing the bean class, the `AnnotatedMethod` representing the producer method, or the `AnnotatedField` representing the producer field.
- `getBean()` returns the `Bean` object that is about to be registered. The `Bean` may implement `Interceptor` or `Decorator`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

```
public interface ProcessSessionBean<X>
        extends ProcessManagedBean<Object> {
    public String getEjbName();
    public SessionBeanType getSessionBeanType();
}
```

- `getEjbName()` returns the EJB name of the session bean.
- `getSessionBeanType()` returns a `javax.enterprise.inject.spi.SessionBeanType` representing the kind of session bean.

```
public enum SessionBeanType { STATELESS, STATEFUL, SINGLETON }
```

```
public interface ProcessManagedBean<X>
        extends ProcessBean<X> {
    public AnnotatedType<X> getAnnotatedBeanClass();
}
```

```
public interface ProcessProducerMethod<T, X>
        extends ProcessBean<X> {
    public AnnotatedMethod<T> getAnnotatedProducerMethod();
    public AnnotatedParameter<T> getAnnotatedDisposedParameter();
}
```

```
public interface ProcessProducerField<T, X>
        extends ProcessBean<X> {
    public AnnotatedField<T> getAnnotatedProducerField();
    public AnnotatedParameter<T> getAnnotatedDisposedParameter();
}
```

If any observer method of a `ProcessBean` event throws an exception, the exception is treated as a definition error by the container.

11.5.12. `ProcessObserverMethod` event

The container must fire an event for each observer method of each enabled bean, before registering the `ObserverMethod` object.

The event object must be of type `javax.enterprise.inject.spi.ProcessObserverMethod<T, X>`, where `T` is the bean class of the bean that declares the observer method and `X` is the observed event type of the observer method.

```
public interface ProcessObserverMethod<T, X> {
    public AnnotatedParameter<T> getAnnotatedEventParameter();
    public ObserverMethod<X> getObserverMethod();
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotatedEventParameter()` returns the `AnnotatedParameter` representing the event parameter.
- `getObserverMethod()` returns the `ObserverMethod` object that will be used by the container to call the observer method.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

If any observer method of a `ProcessObserverMethod` event throws an exception, the exception is treated as a definition error by the container.

Chapter 12. Packaging and deployment

When an application is started, the container must perform *bean discovery*, detect definition errors and deployment problems and raise events that allow portable extensions to integrate with the deployment lifecycle.

Bean discovery is the process of determining:

- The bean archives that exist in the application, and the beans they contain
- Which alternatives, interceptors and decorators are *enabled* for each bean archive
- The *ordering* of enabled interceptors and decorators

Additional beans may be registered programmatically with the container by the application or a portable extension after the automatic bean discovery completes. Portable extensions may even integrate with the process of building the `Bean` object for a bean, to enhance the container's built-in functionality.

12.1. Bean archives

Bean classes of enabled beans must be deployed in *bean archives*.

- A library jar, EJB jar, application client jar or rar archive is a bean archive if it has a file named `beans.xml` in the `META-INF` directory.
- The `WEB-INF/classes` directory of a war is a bean archive if there is a file named `beans.xml` in the `WEB-INF` directory or in the `WEB-INF/classes/META-INF` directory of the war. If a war has a file named `beans.xml` in both the `WEB-INF` directory and in the `WEB-INF/classes/META-INF` directory, then non-portable behavior results. Portable applications must have a `beans.xml` file in only one of the `WEB-INF` or the `WEB-INF/classes/META-INF` directories.
- A directory in the JVM classpath is a bean archive if it has a file named `beans.xml` in the `META-INF` directory.

The container is not required to support application client jar bean archives.

A Java EE container is required by the Java EE specification to support Java EE modules. Other containers may or may not provide support for war, EJB jar or rar bean archives.

The container searches for beans in all bean archives in the application classpath:

- In an application deployed as an ear, the container searches every bean archive bundled with or referenced by the ear, including bean archives bundled with or referenced by wars and EJB jars contained in the ear. The bean archives might be library jars, EJB jars, rars or war `WEB-INF/classes` directories.
- In an application deployed as a war, the container searches every bean archive bundled with or referenced by the war. The bean archives might be library jars or the `WEB-INF/classes` directory.
- In an application deployed as an EJB jar, the container searches the EJB jar, if it is a bean archive, and every bean archive referenced by the EJB jar.
- An embeddable EJB container searches each bean archive in the JVM classpath that is listed in the value of the embeddable container initialization property `javax.ejb.embeddable.modules`, or every bean archive in the JVM classpath if the property is not specified. The bean archives might be directories, library jars or EJB jars.

When searching for beans, the container considers:

- any Java class in any bean archive,
- any `ejb-jar.xml` file in the metadata directory of any EJB bean archive,
- any Java class referenced by the `@New` qualifier of an injection point of another bean, and
- any interceptor or decorator class declared in the `beans.xml` file of any bean archive.

If a bean class is deployed in two different bean archives, non-portable behavior results. Portable applications must deploy each bean class in no more than one bean archive.

If an enum declaring injection points is defined in installed library jars, non-portable behavior results.

Bean archives may contain classes which are not deployed as beans. For example a bean archive might contain message-driven beans.

An extension may be deployed in any archive, including those that are not bean archives.

12.2. Application initialization lifecycle

When an application is started, the container performs the following steps:

- First, the container must search for service providers for the service `javax.enterprise.inject.spi.Extension` defined in Section 11.5, “Container lifecycle events”, instantiate a single instance of each service provider, and search the service provider class for observer methods of initialization events.
- Next, the container must fire an event of type `BeforeBeanDiscovery`, as defined in Section 11.5.1, “BeforeBeanDiscovery event”.
- Next, the container must fire an event of type `ProcessModule` for every bean archive, as defined in Section 11.5.5, “ProcessModule event”.
- Next, the container must perform bean discovery, and abort initialization of the application if any definition errors exist, as defined in Section 2.8, “Problems detected automatically by the container”. Additionally, for every Java EE component class supporting injection that may be instantiated by the container at runtime, the container must create an `InjectionTarget` for the class, as defined in Section 11.2, “The Producer and InjectionTarget interfaces”, and fire an event of type `ProcessInjectionTarget`, as defined in Section 11.5.8, “ProcessInjectionTarget event”.
- Next, the container must fire an event of type `AfterBeanDiscovery`, as defined in Section 11.5.2, “AfterBeanDiscovery event”, and abort initialization of the application if any observer registers a definition error.
- Next, the container must detect deployment problems by validating bean dependencies and specialization and abort initialization of the application if any deployment problems exist, as defined in Section 2.8, “Problems detected automatically by the container”.
- Next, the container must fire an event of type `AfterDeploymentValidation`, as defined in Section 11.5.3, “AfterDeploymentValidation event”, and abort initialization of the application if any observer registers a deployment problem.
- Next, the container must inject any enums declaring injection points.
- Finally, the container begins directing requests to the application.

12.3. Application shutdown lifecycle

When an application is stopped, the container performs the following steps:

- First, the container must destroy all contexts.
- Next, the container must destroy dependent objects injected into enums.
- Finally, the container must fire an event of type `BeforeShutdown`, as defined in Section 11.5.4, “BeforeShutdown event”.

12.4. Bean discovery

The container automatically discovers managed beans (according to the rules of Section 3.1.1, “Which Java classes are managed beans?”) and session beans in bean archives and searches the bean classes for producer methods, producer fields, disposer methods and observer methods.

For each Java class or interface deployed in a bean archive, the container must:

- create an `AnnotatedType` representing the type and fire an event of type `ProcessAnnotatedType`, as defined in Section 11.5.6, “ProcessAnnotatedType event”, and then
- inspect the type metadata to determine if it is a bean or other Java EE component class supporting injection, and then
- detect definition errors by validating the class and its metadata, and then
- if the class is a managed bean, session bean, or other Java EE component class supporting injection, create an `InjectionTarget` for the class, as defined in Section 11.2, “The Producer and InjectionTarget interfaces”, and fire an event of type `ProcessInjectionTarget`, as defined in Section 11.5.8, “ProcessInjectionTarget event”, and then
- if the class is an enabled bean, interceptor or decorator, create a `Bean` object that implements the rules defined in Section 7.3.1, “Lifecycle of managed beans”, Section 7.3.2, “Lifecycle of stateful session beans” or Section 7.3.3, “Lifecycle of stateless session and singleton beans”, and fire an event which is a subtype of `ProcessBean`, as defined in Section 11.5.11, “ProcessBean event”.

For each enabled bean, the container must search the class for producer methods and fields, including resources, and for each producer method or field:

- create a `Producer`, as defined in Section 11.2, “The Producer and InjectionTarget interfaces”, and fire an event of type `ProcessProducer`, as defined in Section 11.5.9, “ProcessProducer event”, and then
- if the producer method or field is enabled, create a `Bean` object that implements the rules defined in Section 7.3.4, “Lifecycle of producer methods”, Section 7.3.5, “Lifecycle of producer fields” or Section 7.3.6, “Lifecycle of resources”, and fire an event which is a subtype of `ProcessBean`, as defined in Section 11.5.11, “ProcessBean event”.

For each enabled bean, the container must search the class for observer methods, and for each observer method:

- create an `ObserverMethod` object, as defined in Section 11.1.3, “The ObserverMethod interface” and fire an event of type `ProcessObserverMethod`, as defined in Section 11.5.12, “ProcessObserverMethod event”.

The container determines which alternatives, interceptors and decorators are enabled, according to the rules defined in Section 5.1.2, “Enabled and disabled beans”, Section 9.4, “Interceptor enablement and ordering” and Section 8.2, “Decorator enablement and ordering”, taking into account any `<alternatives>`, `<interceptors>` and `<decorators>` declarations in the `beans.xml` files, and registers the `Bean` and `ObserverMethod` objects:

- For each enabled bean that is not an interceptor or decorator, the container registers an instance of the `Bean` interface defined in Section 11.1, “The Bean interface”.
- For each enabled interceptor, the container registers an instance of the `Interceptor` interface defined in Section 11.1.2, “The Interceptor interface”.
- For each enabled decorator, the container registers an instance of the `Decorator` interface defined in Section 11.1.1, “The Decorator interface”.
- For each observer method of every enabled bean, the container registers an instance of the `ObserverMethod` interface defined in Section 11.1.3, “The ObserverMethod interface”.

12.5. Integration with Unified EL

The container must provide a Unified EL `ELResolver` to the servlet engine and JSF implementation that resolves bean names using the rules of name resolution defined in Section 5.3, “EL name resolution” and resolving ambiguities according to Section 5.3.1, “Ambiguous EL names”.

- If a name used in an EL expression does not resolve to any bean, the `ELResolver` must return a null value.
- Otherwise, if a name used in an EL expression resolves to exactly one bean, the `ELResolver` must return a contextual instance of the bean, as defined in Section 6.5.2, “Contextual instance of a bean”.