JSR-299: Contexts and Dependency Injection for Java EE

JSR-299 Expert Group

Version: Revised Public Review Draft

Table of Contents

Chapter 1. Architecture

This specification provides a powerful new set of services to Java EE components.

- The lifecycle and interactions of stateful components bound to well-defined *lifecycle contexts*, where the set of contexts is extensible
- A sophisticated, typesafe *dependency injection* mechanism, including a facility for choosing between various components that implement the same Java interface at deployment time
- Integration with the Unified Expression Language (EL), allowing any component to be used directly within a JSF or JSP page
- 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 third-party frameworks to integrate cleanly with the Java EE environment

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

The services defined by this specification allow Java EE components to be bound to lifecycle contexts, to be injected, 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 are bound to contexts as *contextual instances*. Contextual instances may be injected into other objects by the dependency injection service.

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. Any framework 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.

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*. The container may be a Java EE container or an embeddable EJB Lite container.

1.2. Supported environments

An application that takes advantage of these services may be designed to execute in either the Java EE 6, Java EE 5 or Java SE environments. If the application executes in a Java SE environment, the embeddable EJB Lite container provides Java EE services such as transaction management and persistence.

Any Java EE 5 compliant container may support these services. However, certain functionality defined by this specification is optional for Java EE 5 containers. This is the case only when explicitly noted in this specification.

Java EE 6 and embeddable EJB Lite containers must support all functionality defined by this specification.

1.3. Relationship to other specifications

An application developer creates Java EE components such as EJBs, servlets and JavaBeans 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, for example, alternative web presentation technologies.

1.3.1. 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 enironment*. 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 bindings.

1.3.2. 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 EJB instances, even those which are not contextual instances.

1.3.3. 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.

1.3.4. 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 Unified EL name. Thus, a JSF application may take advantage of the sophisticated context and dependency injection model defined by this specification.

1.4. Introductory examples

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

1.4.1. 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 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 {

    @Current Credentials credentials;
    @PersistenceContext EntityManager userDatabase;

    private User user;

    public void login() {

        List<User> results = userDatabase.createQuery(
            "select u from User u where u.username=:username and u.password=:password")
            .setParameter("username", credentials.getUserName())
            .setParameter("password", 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 is a *scope type* that specifies the lifecycle of instances of `Login`.

The `@Current` annotation is a *binding annotation* and causes the `Credentials` bean to be injected into any contextual instance of `Login` created by the container.

The JPA `@PersistenceContext` annotation causes a JPA `EntityManager` to be injected by the container.

The `@LoggedIn` annotation is also a binding annotation. The method annotated `@Produces` is 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 {

    @Current Document document;
    @LoggedIn User user;
    @PersistenceContext EntityManager docDatabase;

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

}
```

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`.

1.4.2. EJB example

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

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

    @Current Credentials credentials;
    @PersistenceContext EntityManager userDatabase;

    private User user;

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

Chapter 2. Bean definition

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 bindings
- A scope
- A deployment type
- Optionally, a bean EL name
- A set of interceptor bindings
- A bean implementation

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 providing additional metadata, or by allowing them to be defaulted by the container. In certain other cases, for example resources defined in Section 3.6, “Resources”, the developer provides only the metadata and the bean implementation is provided by the container.

A bean implementation may be a Java class, an EJB session bean class, a producer method or field or a proxy object for a resource, as specified in Chapter 3, *Bean implementation*. The other attributes of the bean are either:

- declared explicitly by annotating the bean class, or
- defaulted by the container.

The deployment type, bean types and bindings of a bean determine where its instances will be injected by the container.

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 bindings of a bean determine which decorators will be applied at runtime. Interceptors, decorators and interceptor bindings are specified in Appendix A, *Interceptors and decorators*.

A bean implementation may produce or consume events. The event notification facility is specified in Chapter 8, *Events*.

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 7, *Bean lifecycle* and Chapter 6, *Scopes and contexts*,
- scoped resolution by bean type and binding annotation type when injected into a Java-based client, as defined by Section 5.1, “Typesafe resolution algorithm”,
- scoped resolution by name when used in a Unified EL expression, as defined by Section 5.2, “Name resolution algorithm”,
- lifecycle callbacks and automatic injection of other bean instances, specified in Chapter 3, *Bean implementation* and

Chapter 5, *Lookup, dependency injection and EL resolution*,

- method interception, callback interception, and decoration, as defined in Appendix A, *Interceptors and decorators*, and
- event notification, as defined in Chapter 8, *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 three bean types:

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

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

Meanwhile, this session bean has only the local interfaces `BookShop` and `Auditable` 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 set of bean types for a bean are defined in Chapter 3, *Bean implementation*.

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

The bean types of a bean are used by the resolution algorithms defined in Chapter 5, *Lookup, dependency injection and EL 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 an array type. Two array types are considered identical only if the element type is identical.
- A bean type may be a primitive types. Primitive types are considered to be identical to their corresponding wrapper types in `java.lang`.
- A bean type may be a parameterized type with an actual type parameter. Parameterized bean types are considered identical if both the type and the type parameters (if any) are identical.

However, a type declaration containing a type variable or wildcard is not a legal bean type.

If an injection point is declared with a type that is not a legal bean type, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

Note that certain additional restrictions are specified in Section 5.3.1, “Unproxyable bean types” for beans with a normal scope, as defined in Section 6.3, “Normal scopes and pseudo-scopes”.

2.2.2. Typecasting between bean types

A client of a bean may typecast its reference to any instance of the bean to any bean type of the bean. For example, if our managed bean was injected to the following field:

```
@Current Shop<Book> bookShop;
```

Then the following typecast is legal and will not result in an exception:

```
Business biz = (Business) bookShop;
```

Likewise, if our session bean was injected to the following field:

```
@Current BookShop bookShop;
```

Then the following typecast is legal and will not result in an exception:

```
Auditable aud = (Auditable) bookShop;
```

2.3. Bindings

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 that `PaymentProcessor` interface. However, this approach creates a hard dependence between client and implementation—exactly what use of the interface was designed to avoid!

A *binding 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 binding types representing synchronicity and asynchronicity. In Java code, binding types are represented by annotations.

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

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

Finally, binding 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:

```
@Synchronous PaymentProcessor paymentProcessor;
```

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

```
@Asynchronous PaymentProcessor paymentProcessor;
```

The container inspects the binding annotations and type of the injected attribute to determine the bean instance to be injected, according to the resolution algorithm defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

Binding types are also used as event selectors by observers of events, as defined in Chapter 8, *Events*, and to bind decorators to beans, as specified in Section A.5, “Decorators”.

2.3.1. Built-in binding types

Every bean has the built-in binding `@javax.inject.Any`, even if it does not explicitly declare this binding, except for beans with the built-in binding `@javax.inject.New` defined in Section 3.2.4, “Managed beans with the @New binding” and Section 3.3.4, “Session beans with the @New binding”.

If a bean does not explicitly declare a binding, the bean has exactly one additional binding, of type `@javax.inject.Current`. This is called the *default binding*.

The following declarations are equivalent:

```
@Current
public class Order {}
```

```
public class Order {}
```

Both declarations result in a bean with two bindings: `@Any` and `@Current`.

The default binding is also assumed for any injection point that does not explicitly declare a binding. The following declarations, in which the use of the `@Initializer` annotation identifies the constructor parameter as an injection point, are equivalent:

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

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

2.3.2. Defining new binding types

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

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

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

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

A binding type may define annotation members.

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

2.3.3. Declaring the bindings of a bean

A bean's bindings are declared by annotating the bean class or producer method or field with the binding types.

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

```
public class Shop {

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

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

   @Produces @ShoppingCart
   public List<Product> getShoppingCart() { ..... }

}
```

Any bean may declare multiple binding types.

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

2.3.4. Specifying bindings of an injected field

Binding types may be applied to injected fields (see Section 3.8, “Injected fields”) to determine the bean that is injected, according to the typesafe resolution algorithm defined in Section 5.1, “Typesafe resolution algorithm”.

```
@LDAP Authenticator authenticator;
```

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

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

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

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

```
@ShoppingCart List<Product> cart;
```

2.3.5. Specifying bindings of a method or constructor parameter

Binding types may be applied to parameters of producer methods, initializer methods, disposal methods or bean constructors (see Chapter 3, *Bean implementation*) to determine the bean instance that is passed when the method is called by the container. The container uses the typesafe resolution algorithm defined in Section 5.1, “Typesafe resolution algorithm” 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 explictly 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 several standard scope types defined by this specification. 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”. In addition, there is the `@Dependent` pseudo-scope for dependent objects, as defined in Section 6.4, “Dependent pseudo-scope”.

2.4.2. Defining new scope types

A scope type is a Java annotation defined as `@Target({TYPE, METHOD, FIELD})` and `@Retention(RUNTIME)`. All scope types must also specify the `@javax.context.ScopeType` meta-annotation.

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

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

An application or third-party framework might provide a *context* implementation for this custom scope (see Section 9.2.12, “Registering a Context”).

2.4.3. Declaring the bean scope

The bean's scope 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, as defined in Section 10.3, “Problems detected automatically by the container”.

The following examples demonstrate the use of built-in scope types:

```
@RequestScoped
public class ProductList implements DataModel { ... }
```

```
public class Shop {

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

   @Produces @ConversationScoped @ShoppingCart
   public List<Product> getShoppingCart() { ..... }

}
```

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, as defined in Section 10.3, “Problems detected automatically by the container”.

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

2.5. Deployment types

In many applications, there are various implementations of a particular type, and the implementation used at runtime varies between different deployments of the system. Therefore, a developer may associate a particular implementation of a bean type with a certain deployment scenario.

A *deployment type* represents a deployment scenario. Beans may be classified by deployment type, and thereby associated with various deployment scenarios.

Deployment types allow the container to identify which beans should be *enabled* for use in a particular deployment of the system. The deployment type also determines the *precedence* of a bean, used by the resolution algorithms specified in Chapter 5, *Lookup, dependency injection and EL resolution*.

The set of deployment types is extensible.

2.5.1. Built-in deployment types

There are two standard deployment types defined by this specification: `@javax.inject.Production` and `@javax.inject.Standard`.

All standard beans defined by this specification, and provided by the container, are defined using the `@Standard` deployment type. For example, the `Conversation` object defined in Section 6.7.4, “Conversation context lifecycle” and the `BeanManager` object defined in Section 9.2, “The BeanManager object” have this deployment type. No bean may be declared with the `@Standard` deployment type unless explicitly required by this specification.

Application beans may be defined using the `@Production` deployment type.

2.5.2. Defining new deployment types

A deployment type is a Java annotation defined as `@Target({TYPE, METHOD, FIELD})` and `@Retention(RUNTIME)`. All deployment types must also specify the `@javax.inject.DeploymentType` meta-annotation.

Applications and third-party frameworks may define their own deployment types. For example, the following deployment type might identify beans which are used only at a particular site at which the application is deployed:

```
@DeploymentType
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Australian {}
```

This deployment type might be used by a third-party framework that integrates with the container:

```
@DeploymentType
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface DaoFramework {}
```

This deployment type might be used to define mock objects for integration testing:

```
@DeploymentType
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Mock {}
```

2.5.3. Declaring the deployment type of a bean

The deployment type of the bean is declared by annotating the bean class or producer method or field.

A bean class or producer method or field may specify at most one deployment type. If multiple deployment type annotations are specified, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

Open issue: is this too restrictive? We could allow multiple deployment types to be specified, and ignore all but the highest-precedence enabled deployment type.

This bean has the deployment type `@Production`:

```
@Production
public class Order {}
```

This bean has the deployment type `@Mock`:

```
@Mock
public class MockOrder extends Order {}
```

By default, if no deployment type annotation is explicitly specified, a producer method or field inherits the deployment type of the bean in which it is defined.

This producer method has the deployment type `@Production`:

```
@Production
public class Login {

   @Produces
   public User getUser() { ... }

}
```

This producer method has the deployment type `@Australian`:

```
@Production
public class TaxPolicies {

   @Produces @Australian
   public TaxPolicy getAustralianTaxPolicy() { ... }

}
```

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

2.5.4. Default deployment type

When no deployment type is explicitly declared by annotating the bean class or producer method or field, the deployment type is defaulted.

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

- If a bean does not declare any stereotype with a declared default deployment type, then the default deployment type is `@Production`.
- Otherwise, the default deployment type for the bean is the highest-precedence default deployment type declared by any stereotype declared by the bean.

Thus, the following declarations are equivalent:

```
@Production
public class Order {}
```

```
public class Order {}
```

If a bean explicitly declares a deployment type, any default deployment type declared by stereotypes are ignored.

2.5.5. Enabled deployment types

In a particular deployment, only some deployment types are *enabled*. Beans declared with a deployment type that is not enabled are not available to the resolution algorithms defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

The container inspects the deployment type of each bean that exists in a particular deployment (see Section 10.2, “Bean discovery”) to determine whether the bean is *enabled* in this deployment. If the deployment type is enabled, an instance of the bean may be obtained by lookup, injection or EL resolution. Otherwise, the bean is never instantiated by the container.

By default, only the built-in deployment types are enabled. To enable a custom deployment type, a `<Deploy>` element must be included in a `beans.xml` file and the deployment type must be declared using the annotation type name.

```
<Beans>
    <Deploy>
        <Production/>
        <myfwk:DaoFramework/>
        <site:Australian/>
        <myfwk:Mock/>
    </Deploy>
</Beans>
```

If a `<Deploy>` element is specified, the explicitly declared deployment types are enabled, together with `@Standard`, which need not be declared explicitly.

If no `<Deploy>` element is specified in any `beans.xml` file, only the `@Standard` and `@Production` deployment types are enabled.

If the `<Deploy>` element is specified in more than one `beans.xml` document, the container automatically detects the prob-

lem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

2.5.6. Deployment type precedence

In a particular deployment, all enabled deployment types are strongly ordered in terms of *precedence*. The precedence of a deployment type is used by the resolution algorithms defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

If a `<Deploy>` element is specified, the order of the deployment type declarations determines the deployment type precedence. Deployment types which appear later in this list have a higher precedence than deployment types which appear earlier. The `@Standard` deployment type always has the lowest precedence of any deployment type.

If no `<Deploy>` element is specified, the `@Production` deployment type has a higher precedence than the `@Standard` deployment type.

2.6. Bean EL names

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

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

In certain circumstances, multiple beans may share the same name.

Names are used by the EL name resolution algorithm defined in Section 5.1, “Typesafe resolution algorithm”. This allows a bean to be used directly in a JSP or JSF page.

For example, a bean with the name `products` could be used like this:

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

Resources and JMS resources do not have names.

2.6.1. Declaring the bean EL name

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

```
@Named("products")
public class ProductList implements DataModel { ... }
```

If the `@Named` annotation does not specify the `value` member, the default name is assumed.

2.6.2. Default bean EL 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 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 name.

The default name for a bean depends upon the bean implementation. The rules for determining the default name for a bean are defined in Chapter 3, *Bean implementation*.

2.6.3. Beans with no name

If neither `<Named>` nor `@Named` is specified, by the bean or its stereotypes, a bean has no name.

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 deployment type,
- a default scope,
- a restriction upon the bean scope,
- a requirement that the bean implement or extend a certain type, and
- a set of interceptor bindings.

A stereotype may also specify that all beans with the stereotype have defaulted bean EL names.

A bean may declare zero, one or multiple stereotypes.

2.7.1. Defining new stereotypes

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

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

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

A stereotype may not declare any binding annotation. If a stereotype declares a binding annotation, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

2.7.1.1. Declaring the default scope and deployment type for a stereotype

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, as defined in Section 10.3, “Problems detected automatically by the container”.

A stereotype may declare at most one deployment type. If a stereotype declares more than one deployment type, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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

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

Then actions would have scope `@RequestScoped` and deployment type `@Production` unless the scope or deployment type explicitly specified by the bean.

2.7.1.2. Specifying interceptor bindings for a stereotype

A stereotype may declare zero, one or multiple interceptor bindings, as defined in Section A.3.5.2, “Interceptor bindings for stereotypes”.

We may specify interceptor bindings that apply to all actions:

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

2.7.1.3. Specifying name defaulting for a stereotype

A stereotype may declare an empty `@Named` annotation. If a stereotype declares a non-empty `@Named` annotation, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

We may specify that every bean with the stereotype has a defaulted name when a name is not explicitly specified by the bean:

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

2.7.1.4. Restricting bean scopes and types using a stereotype

If all actions are request scoped, we can make this restriction explicit:

```
@RequestScoped
@Secure
@Transactional
@Production
@Stereotype(supportedScopes=RequestScoped.class)
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

We may even require that all actions extend some `ActionBase` class:

```
@RequestScoped
@Secure
@Transactional
@Production
@Stereotype(requiredTypes=ActionBase.class)
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

Scope and type restrictions may not be specified when a stereotype is declared in XML.

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 deployment type and 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. Stereotype restrictions

A stereotype may place certain restrictions upon the beans that declare the stereotype.

If a stereotype declares a `requiredType`, and the bean types do not include the type, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a stereotype explicitly declares a set of scope types using `supportedScopes`, and the bean scope is not in that set, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a bean declares multiple stereotypes, it must satisfy every restriction declared by every declared stereotype.

2.7.4. Built-in stereotypes

The built-in `@Model` stereotype 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 Appendix A, *Interceptors and decorators*.

Chapter 3. Bean implementation

A bean implementation implements the bean types of the bean. The developer must follow certain rules when defining a bean implementation. However, the rules depend upon what kind of bean it is. The container provides built-in support for 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)
- JMS resources (topics and queues)

An application or third-party framework may support other kinds of beans by implementing the interface `Bean` and registering the implementation with the container, as defined in Section 9.2.6, “Bean registration”.

3.1. Restriction upon bean instantiation

Most beans are implemented by an annotated Java class, possibly an EJB bean class, called the *bean class* of the bean. Bean classes are defined in Section 3.2, “Managed beans” and Section 3.3, “Session beans”.

This specification places very few 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 of a bean, instead of letting the container perform instantiation, the resulting instance is not a contextual instance and 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 full control over instantiation of a bean, a *producer method* may be used. A producer method is just an annotated method of another bean that is invoked by the container to instantiate the bean. Producer methods are defined in Section 3.4, “Producer methods”. However, a similar restriction exists for producer methods: if the application calls the producer method directly, instead of letting the container call it, the returned object is not a contextual instance and the capabilities listed in Section 2.1, “Functionality provided by the container to the bean” will not be available to the returned object.

3.2. 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, as defined in Section 10.3, “Problems detected automatically by the container”.

If a managed bean has a public field, it must have scope `@Dependent`. If a managed bean with a public field declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

3.2.1. 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 parameterized type.

- 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 has an appropriate constructor—either:
  - the class has a constructor with no parameters, or
  - the class declares a constructor annotated `@Initializer`.

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

3.2.2. Bean types of a managed bean

The 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 5.3.1, “Unproxyable bean types”.

3.2.3. 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 { ... }
```

A bean class may also specify a scope, name, deployment type, stereotypes and/or bindings:

```
@ConversationScoped @Current
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.

3.2.4. Managed beans with the `@New` binding

Every class that satisfies the requirements of Section 3.2.1, “Which Java classes are managed beans?” is a bean, with scope, deployment type and bindings defined using annotations.

Additionally, for each such managed bean, a second managed bean exists which:

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

However, this second bean:

- has scope `@Dependent`,
- has deployment type `@Standard`,
- has `@javax.inject.New` as the only binding,
- has no bean EL name,
- has no stereotypes, and
- has no observer methods, producer methods or fields or disposal methods.

3.2.5. Bean constructors

When the container instantiates a managed bean, it calls the *bean constructor*. The bean constructor is a 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.

3.2.5.1. Declaring a bean constructor

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

```
@SessionScoped
public class ShoppingCart {

   private User customer;

   @Initializer
   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;

   @Initializer
   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 managed bean does not explicitly declare a constructor using `@Initializer`, the constructor that accepts no parameters is the bean constructor.

If a managed bean has more than one constructor annotated `@Initializer`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a bean constructor has a parameter annotated `@Disposes`, or `@Observes`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

3.2.5.2. Bean constructor parameters

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

3.2.6. 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, as defined in Section 10.3, “Problems detected automatically by the container”.

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

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

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

3.2.7. Default 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 EL name is `productList`.

3.3. Session beans

An *session bean* is a bean that is implemented by a session bean with an EJB 3.x client view. The basic lifecycle and semantics of an EJB session bean 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, as defined in Section 10.3, “Problems detected automatically by the container”. 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, as defined in Section 10.3, “Problems detected automatically by the container”.

3.3.1. 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 an instance of the session bean.
- Otherwise, the application *may not* directly call any EJB remove method of any instance of the session bean.

If the application directly calls an EJB remove method of an 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 an 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 7.2, “Lifecycle of stateful session beans”.

3.3.2. Bean types of a session bean

The set of bean types for a session bean contains all local interfaces of the bean that do not have wildcard type parameters or type variables and their superinterfaces. If the EJB has a bean class local view and the bean class is not a parameterized type, the 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.

3.3.3. Declaring a session bean

A session bean does not require any special annotations. The following EJBs are beans:

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

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

A bean class may also specify a scope, name, deployment type, stereotypes and/or bindings:

```
@ConversationScoped @Stateful @Current @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 { ... }
```

3.3.4. Session beans with the `@New` binding

Every session bean with an EJB 3.x client view is a bean, with scope, deployment type and bindings defined using annotations.

Additionally, for each such session bean, a second bean exists which:

- has the same bean class,
- has the initializer methods and injected fields defined by annotations, and
- has the same interceptor bindings defined by annotations.

However, this second bean:

- has scope `@Dependent`,
- has deployment type `@Standard`,
- has `@javax.inject.New` as the only binding,
- has no bean EL name,

- has no stereotypes, and
- has no observer methods, producer methods or fields or disposal methods.

3.3.5. 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, as defined in Section 10.3, “Problems detected automatically by the container”.

Furthermore:

- X must support all local interfaces supported by Y, and
- if Y supports a bean-class local view, X must also support a bean-class local view.

Otherwise, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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

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

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

3.4. 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 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 5.3, “Client proxies”.

If the producer method return type is a parameterized type, it must specify actual type parameters for each type parameter. If a producer method return type contains a wildcard type parameter or type variable, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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.

3.4.1. 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 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 set of bean types contains exactly two types: the method return type and `java.lang.Object`.
- If the return type is a class, the 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 5.3.1, “Unproxyable bean types”.

3.4.2. Declaring a producer method

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

```
public class Shop {

   @Produces PaymentProcessor getPaymentProcessor() { ... }

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

}
```

A producer method may also specify scope, name, deployment type, stereotypes and/or bindings.

```
public class Shop {

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

}
```

If a producer method is annotated `@Initializer`, has a parameter annotated `@Disposes`, or has a parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a non-static method of a session bean class is annotated `@Produces`, and the method is not a business method of the EJB, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

3.4.3. Producer method parameters

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

```
public class OrderFactory {

   @Produces @ConversationScoped
   public Order createCurrentOrder(@New Order order, @Selected Product product)
   {
       order.setProduct(product);
       return order;
   }

}
```

3.4.4. 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 inherits all bindings of Y, and
- if Y has a name, X has the same name as Y.

We say that X *directly specializes* Y, and we can be certain that Y will never be called by the container if X is enabled.

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, as defined in Section 10.3, “Problems detected automatically by the container”.

For example:

```
@Mock
public class MockShop extends Shop {

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

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

   ...

}
```

3.4.5. Disposal methods

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

A disposal method must be a method of a managed bean class or session bean class. A disposal method may be either static or non-static. If the bean is a session bean, the disposal method must be a business method of the EJB or a static method of the bean class.

A bean may declare multiple disposal methods.

3.4.6. Disposed parameter of a disposal method

Each disposal method must have exactly one *disposed parameter*, of the same type as the corresponding producer method return type. When searching for disposal methods for a producer method, the container considers the type and bindings of the disposed parameter. If a disposed parameter resolves to a producer method according to the typesafe resolution algorithm, the container must call this method when destroying an instance returned by that producer method.

If the disposed parameter does not resolve to any producer method according to the typesafe resolution algorithm, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

3.4.7. Declaring a disposal method

A disposal method may be declared by annotating a parameter `@javax.inject.Disposes`. That parameter is 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();
    }

}
```

If a method has more than one parameter annotated `@Disposes`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a disposal method is annotated `@Produces`, or `@Initializer` or has a parameter annotated `@Observes`, the container

automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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 EJB, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

3.4.8. Disposal method parameters

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

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

3.4.9. Disposal method resolution

When searching for disposal methods for a producer method, the container searches for disposal methods which satisfy the following rules:

- The disposal method must be declared by an enabled bean.
- The disposed parameter must resolve to the producer method, according to the typesafe resolution algorithm.

If there are multiple disposal methods for a producer method, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

3.4.10. Default 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`:

```
public class Shop {

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

}
```

This producer method is named `paymentProcessor`:

```
public class Shop {

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

}
```

3.5. Producer fields

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

A producer field must be a field of a managed bean class or session bean class. A producer field may be either static or non-static.

If a producer field sometimes contains a null value when accessed, then the producer field must have scope `@Dependent`. If a producer method 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 5.3, “Client proxies”.

If the producer field return type is a parameterized type, it must specify actual type parameters for each type parameter. If a producer field return type contains a wildcard type parameter or type variable, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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.

3.5.1. 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 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 set of bean types contains exactly two types: the field type and `java.lang.Object`.
- If the field type is a class, the 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 5.3.1, “Unproxyable bean types”.

3.5.2. Declaring a producer field

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

```
public class Shop {

   @Produces PaymentProcessor paymentProcessor = ....;

   @Produces List<Product> products = ....;

}
```

A producer field may also specify scope, name, deployment type, stereotypes and/or bindings.

```
public class Shop {

   @Produces @ApplicationScoped @Catalog @Named("catalog")
   List<Product> products = ....;

}
```

3.5.3. Default name for a producer field

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

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

```
public class Shop {

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

}
```

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.

```
@CustomerDatabase Datasource customerData;
```

```
@CustomerDatabase EntityManager customerDatabaseEntityManager;
```

```
@CustomerDatabase EntityManagerFactory customerDatabaseEntityManagerFactory;
```

```
@Current PaymentService remotePaymentService;
```

A resource always has scope `@Dependent`.

A resource may not declare a bean EL 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.

- 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.

```
@WebServiceRef(name="java:app/service/PaymentService",
               wsdlLocation="http://theirdomain.com/services/PaymentService.wsdl")
PaymentService paymentService;
```

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

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

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

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

The bean type, bindings and deployment type of the resource are determined by the producer field declaration.

3.7. JMS resources

Beans that send JMS messages must interact with at least two different objects defined by the JMS API:

- to send a message to a queue, the bean must interact with a `QueueSession` and the `QueueSender`, or
- to send a message to a topic, the bean must interact with a `TopicSession` and the `TopicPublisher`.

A *JMS resource* is a bean that represents a reference to a JMS queue or topic in the Java EE component environment.

- For a queue, the `Queue`, `QueueConnection`, `QueueSession`, `QueueReceiver` and/or `QueueSender` may be injected.
- For a topic, the `Topic`, `TopicConnection`, `TopicSession`, `TopicSubscriber` and/or `TopicPublisher` may be injected.

The lifecycles of the injected objects are managed by the container, and therefore the application need not explicitly `close()` any injected JMS object. If the application calls `close()` on an instance of a JMS resource, an `UnsupportedOperationException` is thrown by the container.

For example:

```
@PaymentProcessor QueueSender paymentSender;
@PaymentProcessor QueueSession paymentSession;
```

```
public void sendMessage() {
    MapMessage msg = paymentSession.createMapMessage();
    ...
    paymentSender.send(msg);
}
```

```
@Prices TopicPublisher pricePublisher;
@Prices TopicSession priceSession;

public void sendMessage(String price) {
    pricePublisher.send( priceSession.createTextMessage(price) );
}
```

A JMS resource always has scope `@Dependent`.

A JMS resource may not declare a bean EL name.

3.7.1. Bean types of a JMS resource

The bean types of a JMS resource depend upon whether it represents a queue or topic.

- If the JMS resource represents a queue, the bean types are `Queue`, `QueueConnection`, `QueueSession` and `QueueSender`.
- If the JMS resource represents a topic, the bean types are `Topic`, `TopicConnection`, `TopicSession` and `TopicPublisher`.

3.7.2. Declaring a JMS resource

A JMS resource may be declared by specifying a `@Resource` annotation as part of a producer field declaration of type `Topic` or `Queue`.

```
@Resource(name="java:global/env/jms/PaymentQueue")
@Produces @PaymentProcessor Queue paymentQueue;
```

```
@Resource(name="java:global/env/jms/Prices")
@Produces @Prices Topic pricesTopic;
```

The bindings and deployment type of the resource are determined by the producer field declaration.

3.8. Injected fields

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

As defined in Section 5.4, “Dependency injection”, injected fields are initialized by the container before initializer methods are called, and before the `@PostConstruct` callback occurs.

Open issue: are injected fields allowed to be declared transient? If so, should they be reinjected after deserialization (activation)?

If a field is a producer field or a decorator delegate attribute, it is not an injected field.

3.8.1. Declaring an injected field

An injected field may be declared by annotating the field with any binding type.

```
@ConversationScoped
public class Order {

   @Selected Product product;
   @Current User customer;

}
```

3.9. Initializer methods

An *initializer method* is a non-static method of a bean class, 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.

As defined in Section 5.4, “Dependency injection”, initializer methods are called by the container after injected fields are initialized, and before the `@PostConstruct` callback occurs.

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.

3.9.1. Declaring an initializer method

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

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

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

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

}
```

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, as defined in Section 10.3, “Problems detected automatically by the container”.

3.9.2. Initializer method parameters

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

3.10. The default binding at injection points

If an injection point declares no binding, the default binding `@Current` is assumed.

The following are equivalent:

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

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

}
```

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

```
   private Product product;
   private User customer;

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

}
```

The following definitions are equivalent:

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

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

}
```

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

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

}
```

Chapter 4. Inheritance and specialization

The implementation of a bean may by extended by the implementation of a second bean. There are two possible scenarios in which this situation occurs:

- The second bean *specializes* the first bean in a particular deployment scenario. In that deployment, 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 producer, disposal and observer methods of the first bean are not inherited by the second bean.

The bean developer may explicitly specify that the second bean specializes the first through use of an annotation. In the case of specialization, the specialized bean receives all invocations, including producer, disposal and observer method invocations that would have been received by the first bean. In a particular deployment, there may be only one bean that fulfills the specific role. The specialized bean inherits, and may not override, the bindings and name of the first bean.

However, in both cases, the inheritance of type-level metadata is controlled via use of the Java `@Inherited` meta-annotation.

4.1. Inheritance of type-level metadata

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

- If X is annotated with a binding 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 the same as that defined in 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 that defined in the Java Language Specification.)

- If X is annotated with a deployment 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 deployment type.

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

Scope types and deployment types explicitly declared by and inherited from the class X take precedence over default scopes and deployment types declared by stereotypes.

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

- scope types should be declared `@Inherited`,
- binding types should not be declared `@Inherited`,
- deployment 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, binding types, deployment types and interceptor binding types defined by this specification adhere to these recommendations.

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

Note that the `@Named` annotation is not declared `@Inherited` and bean EL 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 simple or session bean Y.

- If X declares an injected field `x` then Y inherits `x`.
- If X declares an initializer method, `@PostConstruct` method 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()`.
- 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()`.
- If X declares a non-static producer, disposal, or observer method `x()` then Y does not inherit this method unless Y is explicitly declared to specialize X.
- If X declares a non-static producer field `x` then Y does not inherit this field unless Y is explicitly declared to specialize X.
- If Y is a decorator and X declares a delegate attribute `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 defines a delegate attribute.

4.3. Specialization

If two beans both support a certain bean type, and share at least one binding, then they are both eligible for injection to any injection point with that declared type and binding. The container will choose the bean with the highest priority enabled deployment type.

Consider the following beans:

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

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

Suppose that the deployment type `@Mock` is enabled:

```
<Beans>
    <Deploy>
        <Production/>
        <myfwk:Mock/>
    </Deploy>
</Beans>
```

Then the following attribute will receive an instance of `MockAsynchronousService`:

```
@Current Service service;
```

However, if the bean with the lower priority deployment type declares a binding that is not declared by the bean with the higher priority deployment type, then the bean with the higher priority deployment type will not be eligible for injection to an injection point with that binding.

Therefore, the following attribute will receive an instance of `AsynchronousService` even though the deployment type `@Mock` is enabled:

```
@Current @Asynchronous Service service;
```

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

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

To help prevent developer error, the first bean may:

- directly extend the bean class of the lower-priority bean, or
- directly override the lower-priority producer method, in the case of a producer method bean, and then

explicitly declare that it *specializes* the lower-priority bean.

4.3.1. Using specialization

A bean may declare that it specializes a lower-priority bean using the `@Specializes` annotation.

Then the first bean will inherit the bindings and name of the lower-priority bean:

- The bindings of a bean X that specializes a lower-priority bean Y include all bindings of Y, together with all bindings declared explicitly by X.
- If a bean X specializes a lower-priority bean Y with a name, the name of X is the same as the name of Y. If X declares a name explicitly, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

For example, the following bean would have the inherited bindings `@Current` and `@Asynchronous`:

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

If `AsynchronousService` declared a name:

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

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

When an enabled bean specializes a lower-priority bean, we can be certain that the lower-priority bean is never instantiated or called by the container. Even if the lower-priority bean defines a producer method, the method will be called upon an instance of the first bean.

Specialization applies only to managed beans, as defined in Section 3.2.6, “Specializing a managed bean”, session beans, as defined in Section 3.3.5, “Specializing a session bean” and producer methods, as defined in Section 3.4.4, “Specializing a producer method”.

4.3.2. Direct and indirect specialization

The `@javax.inject.Specializes` annotation is used to indicate that one bean *directly specializes* another bean.

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.

If X specializes Y but does not directly specialize Y, we say that X *indirectly specializes* Y.

If, in a particular deployment, a bean with a certain bean type and set of bindings is not specialized by any other enabled bean, we call it the *most specialized bean* for that combination of type and bindings in that deployment.

Any non-static producer methods (see Section 3.4, “Producer methods”), producer fields (see Section 3.5, “Producer fields”), disposal methods (see Section 3.4.5, “Disposal methods”) or observer methods (see Section 8.6, “Observer methods”) of any bean are invoked upon an instance of the most specialized enabled bean that specializes the bean, as defined by Section 7.4, “Lifecycle of producer methods”, Section 7.5, “Lifecycle of producer fields” and Section 8.4, “Observer notification”.

4.3.3. Inconsistent specialization

If, in a particular deployment, either

- some enabled bean X specializes another enabled bean Y and X does not have a higher precedence than Y, or
- more than one enabled bean directly specializes the same bean

we say that *inconsistent specialization* exists. The container automatically detects inconsistent specialization and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

Chapter 5. Lookup, dependency injection and EL resolution

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

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

Contextual references may also be obtained by evaluating EL expressions or by programmatic lookup.

In general, a bean type or bean EL name does not uniquely identify a bean. When resolving a bean at an injection point, the container considers bean type, bindings and deployment type precedence. When resolving a bean name in EL, the container considers name and deployment type precedence. This allows bean developers to decouple type from implementation.

The container is required to support circularities in the bean dependency graph.

5.1. Typesafe resolution algorithm

The process of matching a bean to an injection point is called *typesafe resolution*. The container considers bean type, bindings, and deployment precedence when resolving a bean to be injected to an injection point. The type and bindings of the injection point are called the *required type* and *required bindings*.

Typesafe resolution usually occurs at container deployment time, allowing the container to warn the user if any enabled beans have unsatisfied or ambiguous dependencies.

The following algorithm must be used by the container when resolving a bean by type:

- First, the container identifies the set of *matching* enabled beans which have the required bean type. For this purpose, primitive types are considered to be identical to their corresponding wrapper types in `java.lang`, array types are considered identical only if their element types are identical and parameterized types are considered identical only if both the type and all type parameters are identical.
- Next, the container considers the required bindings. If no bindings were explicitly specified, the container assumes the binding `@Current`. The container narrows the set of matching beans to just those where for each required binding, the bean declares a matching binding with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding` (see Section 5.1.3, “Binding annotations with members”).
- Next, the container examines the deployment types of the matching beans, as defined in Section 2.5.6, “Deployment type precedence”, and returns the set of beans with the highest precedence deployment type that occurs in the set. If there are no matching beans, an empty set is returned.

5.1.1. Unsatisfied and ambiguous dependencies

An *unsatisfied dependency* exists at an injection point when no enabled bean has the bean type and bindings declared by the injection point.

An *ambiguous dependency* exists at an injection point when in the set of enabled beans with the bean type and bindings declared by the injection point there exists no unique bean with a higher precedence than all other beans in the set.

The container must validate all injection points of all enabled beans at deployment time to ensure that there are no unsatisfied or ambiguous dependencies. If an unsatisfied or ambiguous dependency exists, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

The method `Bean.getInjectionPoints()` may be used to determine the dependencies of a bean.

5.1.2. 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 bean that may be null, such as a producer method with a non-primitive return type or a producer field with a non-primitive type, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

The method `Bean.isNullable()` may be used to detect if a bean has null values.

5.1.3. Binding annotations with members

According to the algorithm above, binding types with members are supported:

```
@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:

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

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

```
@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.

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

Array-valued or annotation-valued members of a binding type must be annotated `@NonBinding`. If an array-valued or annotation-valued member of a binding type is not annotated `@NonBinding`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

5.1.4. Multiple bindings

According to the algorithm above, a bean class or producer method or field may declare multiple bindings:

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

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

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

```
@Synchronous PaymentProcessor paymentProcessor;
```

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

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

5.2. Name resolution algorithm

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 bean EL names.

The following algorithm must be used by the container when resolving a bean by name:

- The container identifies the set of *matching* enabled beans which have the given name.
- Next, the container examines the deployment types of the matching beans, as defined in Section 2.5.6, “Deployment type precedence”, and returns the set of beans with the highest precedence deployment type that occurs in the set. If there are no matching beans, an empty set is returned.

The name resolution algorithm usually occurs at runtime.

5.3. 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 all 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 are serializable, even when the bean itself is not. 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, “Passivating scopes and serialization”. (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`.

All client proxies must be serializable.

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.)

5.3.1. Unproxyable bean types

Certain legal bean types cannot be proxied by the container:

- classes without a non-private constructor with no parameters,
- classes which are declared final or have final methods,
- primitive types,
- and array types.

If an injection point whose declared type cannot be proxied by the container resolves to a bean with a normal scope, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

5.3.2. 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.

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.

5.4. 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.

In a Java EE 5 environment, the container is not required to support injection for non-contextual objects.

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

5.4.1. Injectable references

To obtain an *injectable reference* for an injection point, the container must:

- Identify a bean according to the rules defined in Section 5.1, “Typesafe resolution algorithm”.
- If typesafe resolution resulted in an empty set, throw an `UnsatisfiedResolutionException`.
- Otherwise, if typesafe resolution resulted in more than one bean, throw an `AmbiguousResolutionException`.
- Otherwise, if the resulting bean has a normal scope and the type cannot be proxied by the container, as defined in Section 5.3.1, “Unproxyable bean types”, throw an `UnproxyableResolutionException`.
- Otherwise, obtain a contextual reference for this bean according to Section 6.5.3, “Contextual reference for a bean”.

Normally, typesafe resolution results in exactly one bean, since the container validated dependencies at deployment time.

For certain combinations of scopes, the container is permitted to optimize the algorithm above:

- 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 rule of *injected reference validity*.

5.4.2. Injected reference validity

Injected references to a bean are *valid* until the object into which they were injected is destroyed. The application should not invoke a method of an invalid reference. If the application invokes a method of an injected reference after the object into which it was injected has been destroyed, the behavior is undefined.

5.4.3. Injection using the bean constructor

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

5.4.4. 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, the container must perform the following steps after injection of Java EE component environment resources has been performed and before the `@PostConstruct` callback occurs and before the servlet `init()` method is called:

- First, the container initializes the values of all injected fields. The container sets the value of each injected field to an injectable reference.
- Next, the container calls all initializer methods, passing an injectable reference to each parameter.
- Finally, if the component supports interception, the container builds the interceptor and decorator stacks for the instance as defined in Section A.3.11, “Interceptor stack creation” and Section A.5.7, “Decorator stack creation” and binds them to the instance.

5.4.5. Destruction of dependent objects

When the container destroys an instance of a bean or of any Java EE component class supporting injection, the container destroys all dependent objects, as defined in Section 6.4.3, “Dependent object destruction”, after the `@PreDestroy` callback completes and after the servlet `destroy()` method is called.

5.4.6. Invocation of producer or disposal methods

When the container calls a producer or disposal 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:
    - Determine the most specialized bean that specializes the bean which declares the method.
    - Obtain a contextual instance of the most specialized bean, as defined by Section 6.5.2, “Contextual instance of a bean”.
    - Invoke the method upon this instance.

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 6.4.3, “Dependent object destruction”.

5.4.7. 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:
  - Determine the most specialized bean that specializes the bean which declares the producer field.
  - Obtain an contextual instance of the most specialized bean, as defined by Section 6.5.2, “Contextual instance of a bean”.
  - Access the field value of this instance.

5.4.8. Invocation of observer methods

When the container calls an observer method (defined in Section 8.6, “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:
  - Determine the most specialized bean that specializes the bean which declares the observer method.
  - Obtain a contextual instance of the bean 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, without creating a new contextual instance.
  - Invoke the observer method on the resulting instance, if any.

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 6.4.3, “Dependent object destruction”.

5.4.9. Injection point metadata

The interface `javax.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, disposal method or observer method.

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

- 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.
- The `getType()` and `getBindings()` methods return the declared type and bindings of the injection point.
- 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.
- The `getAnnotated()` method returns an instance of `javax.inject.spi.AnnotatedField` or `javax.inject.spi.AnnotatedParameter`, depending upon whether the injection point is an injected field or a constructor/method parameter.

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 `Loggers`. 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 deployment type `@Standard`, scope `@Dependent`, bean type `InjectionPoint` and binding `@Current`, allowing dependent objects, as defined in Section 6.4.2, “Dependent objects”, to obtain information about the injection point to which they belong.

If a bean that declares any scope other than `@Dependent` has an injection point of type `InjectionPoint` and binding `@Current`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If an object that is not a bean has an injection point of type `InjectionPoint` and binding `@Current`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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

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

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

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

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

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

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

Or, the `@Any` binding may be used, allowing the application to specify bindings dynamically:

```
@Any Instance<PaymentProcessor> anyPaymentProcessor;
...
Annotation binding = synchronously ? new SynchronousBinding() : new AsynchronousBinding();
PaymentProcessor pp = anyPaymentProcessor.select(binding).get().process(payment);
```

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

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

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

5.5.1. The `Instance` interface

The `Instance` interface provides a method for obtaining instances of beans of a specific type, and inherits the ability to iterate beans of a specific type from `java.lang.Iterable`:

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

    public T get();

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

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

}
```

The `select()` method of the provided implementation of `Instance` returns a child `Instance` for a subtype of the bean type and additional bindings. If no subtype is given, the bean type is the same as the parent.

If a parameterized type with a type parameter or wildcard is passed to `select()`, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `select()`, a `DuplicateBindingTypeException` is thrown.

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

The `get()` method of the provided implementation of `Instance` must:

- Identify the bean by calling `BeanManager.getBeans()`, passing the type and bindings specified at the injection point. A child `Instance` passes the bean subtype and the additional bindings, along with the bindings of its parent.
- If `getBeans()` did not return a bean, throw an `UnsatisfiedResolutionException`.
- Otherwise, if `getBeans()` returned more than one bean, throw an `AmbiguousResolutionException`.
- Otherwise, obtain a contextual reference for the bean by calling `BeanManager.getReference()`, passing the `Bean` object representing the bean, and return it.

The `iterate()` method of the provided implementation of `Instance` must:

- Identify the set of beans by calling `BeanManager.getBeans()`, passing the type and bindings specified at the injection point. A child `Instance` passes the subtype, the bindings specified at the injection point and the additional bindings.
- Return an `Iterator`, that iterates over the set of contextual references for the resulting beans by calling `BeanManager.getReference()`, passing the `Bean` object representing the current bean.

5.5.2. The built-in `Instance`

The container must provide a built-in bean with:

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

5.5.3. Using `AnnotationLiteral`

When the application calls `select()`, it may pass instances of binding annotation types. The helper class `javax.inject.AnnotationLiteral` makes it easier to implement annotation types:

```
public class SynchronousBinding
          extends AnnotationLiteral<Synchronous>
          implements Synchronous {}
```

```
public abstract class PayByBinding
          extends AnnotationLiteral<PayBy>
          implements PayBy {}
```

Then the application may easily instantiate instances of the binding type:

```
PaymentProcessor pp = paymentProcessor.get( new SynchronousBinding(),
        new PayByBinding() { public PaymentMethod value() { return CHEQUE; } });
```

5.6. Integration with Unified EL

The container must provide a Unified EL `ELResolver` to the servlet engine and JSF implementation that resolves bean EL names.

When this resolver is called with a null base object, it must:

- Identify the bean using the name resolution algorithm defined in Section 5.2, “Name resolution algorithm”.
- If name resolution resulted in an empty set, return a null value.
- Otherwise, if resolution resulted in more than one bean, throw an `AmbiguousResolutionException`.
- Otherwise, if exactly one bean results, obtain a contextual instance of the bean, as defined in Section 6.5.2, “Contextual instance of a bean”, and return it.

For each distinct name that appears in the EL expression, the resolver must be called at most once. Even if a name appears more than once in the same expression, the container may not call the resolver multiple times with that name. This restriction ensures that there is a unique instance of each bean with scope `@Dependent` in any EL evaluation.

Open issue: qualified names are supported. The `ELResolver` implements support for qualified names in Unified EL. How exactly does this work?

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 `javax.context.Contextual` interface defines operations to create and destroy contextual instances of a certain type:

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

Any implementation of `Contextual` is called a *contextual type*.

In particular, the `Bean` interface defined in Section 9.1, “The Bean interface” extends `Contextual`, so all beans are contextual types.

The container and third party frameworks 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. Instance creation

The `Contextual.create()` method is responsible for creating new contextual instances of the type.

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

The interface `javax.context.CreationalContext` provides an operation that allows the `create()` method to register an incompletely initialized contextual instance with the container. A contextual instance is considered *incompletely initialized* until the `create()` method returns the instance.

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

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

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

6.1.2. Instance destruction

The `Contextual.destroy()` method is responsible for destroying instances of the type. In particular, it is responsible for destroying all dependent objects of an instance, as defined in Section 6.4.3, “Dependent object destruction”.

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

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

6.2. The `Context` interface

The `javax.context.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 third party frameworks. It should not be called directly by the application.

```
public interface Context {

    public Class<? extends Annotation> getScopeType();

    public <T> T get(Contextual<T> bean);
    public <T> T get(Contextual<T> bean, CreationalContext<T> creationalContext);

    boolean isActive();

}
```

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()` 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.

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.

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 enabled 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 in-

stances 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 typesafe resolution algorithm. 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 pseudo-scopes must be explicitly declared `@ScopeType(normal=false)`, 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 `@javax.context.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 injected instance of the bean is bound to the lifecycle of the instance into which it is injected.
- Any instance of the bean that is used to evaluate a Unified EL expression exists to service that evaluation only.
- Any instance of the bean that receives a producer method, producer field, disposal method or observer method invocation exists to service that invocation only.

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.

6.4.1. Dependent scope lifecycle

The `@Dependent` scope is inactive except:

- when an instance of a bean with scope `@Dependent` is created by the container to receive a producer method, producer field, disposal method or observer method invocation, or
- while a Unified EL expression is evaluated, or
- while an observer method is invoked, or
- when the container is creating or destroying a contextual instance of a bean, injecting its dependencies, invoking its observer methods, or invoking its `@PostConstruct` or `@PreDestroy` callback, or
- when the container is injecting dependencies or invoking the `@PostConstruct` or `@PreDestroy` callback of a Java EE component class supporting injection, or
- when `Instance.get()` or `BeanManager.getReference()` is invoked upon an instance of `Instance` or `BeanManager` injected by the container into a bean or other Java EE component class supporting injection.

The `@Dependent` scope is not active when `Instance.get()` or `BeanManager.getReference()` is invoked upon an instance of `Instance` or `BeanManager` that was not injected by the container.

6.4.2. 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 interceptors or decorators with scope `@Dependent` are dependent objects of the bean instance they intercept or decorate.
- An instance of a bean with scope `@Dependent` injected into a field, bean constructor, initializer method or observer 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 or disposal method is a dependent object of the producer method bean instance that is being produced or disposed.
- An instance of a bean with scope `@Dependent` obtained by direct invocation of `BeanManager` or `Instance` during invocation of a bean constructor, initializer method, observer method, `@PostConstruct` or `@PreDestroy` callback is a dependent object of the bean or Java EE component class instance upon which the method is being invoked.
- An instance of a bean with scope `@Dependent` obtained by direct invocation of `BeanManager` or `Instance` during invocation of a producer method or disposal method is a dependent object of the producer method bean instance that is being produced or disposed.
- Otherwise, an instance of a bean with scope `@Dependent` obtained by direct invocation of an instance of `Instance` or `BeanManager` that was injected by the container into a bean or Java EE component class instance is a dependent object of the bean or Java EE component class instance.

6.4.3. Dependent object destruction

The container is responsible for destroying `@Dependent` scoped contextual instances by passing them to the `Contextual.destroy()` method.

The container must:

- destroy all dependent objects of a contextual bean instance when the instance is destroyed,
- destroy all dependent objects of a non-contextual instance of a bean or instance of other Java EE component class when the instance is destroyed,
- destroy all `@Dependent` scoped contextual instances created during an EL expression evaluation when the evaluation completes, and
- destroy any `@Dependent` scoped contextual instance created to receive a producer method, producer field, disposal method or observer method invocation when the invocation 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).

6.5. 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`.

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:

- 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 without passing any instance of `CreationalContext`.

6.5.3. Contextual reference for a bean

From time to time, the container must obtain a *contextual reference* for a bean.

- If the bean has a normal scope, the contextual reference of the bean is a client proxy created by the container.
- Otherwise, if the bean has a pseudo-scope, the container must obtain a contextual instance of the bean.

The container must ensure that every injection point of type `InjectionPoint` and binding `@Current` 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.

The container is required to ensure that any contextual reference for a bean may be cast to any bean type of the bean.

6.5.4. Contextual reference validity

Contextual reference of a bean are *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 depends upon whether the scope of the injected 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.

6.6. Passivating scopes and serialization

A *passivating scope* requires that instances of beans with that scope be serializable, so that their state may be stored to disk when the scope becomes inactive. The process of storing the state of contextual instances belonging to a scope that is about to become inactive to disk is called *context passivation*. Passivating scopes must be explicitly declared `@ScopeType(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.

The container must validate that every bean declared with a passivating scope truly is serializable:

- EJB local objects are serializable. Therefore, a session bean may declare any passivating scope.
- Managed beans are not required to be serializable. If a managed bean declares a passivating scope, and the bean class is not serializable, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.
- If a producer method or field declares a passivating scope and returns a non-serializable object at runtime, an `IllegalProductException` is thrown by the container.

The built-in session and conversation scopes are passivating. No other built-in scope is passivating.

A contextual instance of a bean may be serialized under one of two circumstances:

- the bean declares a passivating scope, and context passivation occurs, or
- the bean is an EJB stateful session bean, and it is passivated by the EJB container.

In either case, any non-transient field that holds a reference to another bean must be serialized along with the bean that is being serialized. Therefore, the reference must be to a serializable type.

Client proxies are serializable. Therefore, any reference to a bean which declares a normal scope is serializable. On the other hand, dependent objects (including interceptors and decorators with scope `@Dependent`) of a stateful session bean or of a bean with a passivating scope must be serialized and deserialized along with their owner:

- EJB local objects are serializable. Therefore, any reference to a session bean of scope `@Dependent` is serializable.
- A managed bean of scope `@Dependent` may or may not be serializable. If a managed bean of scope `@Dependent` and a non-serializable bean class is injected into a stateful session bean, into a non-transient field, bean constructor parameter or initializer method parameter of a bean which declares a passivating scope, or into a parameter of a producer method which declares a passivating scope, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.
- If a producer method or field of scope `@Dependent` returns a non-serializable object for injection into a stateful session bean, into a non-transient field, bean constructor parameter or initializer method parameter of a bean which declares a passivating scope, or into a parameter of a producer method which declares a passivating scope, an `IllegalProductException` is thrown by the container.
- The container must guarantee that JMS resource proxy objects are serializable.

The method `Bean.isSerializable()` may be used to detect if a bean is serializable.

6.7. Context management for built-in scopes

The container provides an implementation of the `Context` interface for each of the built-in scopes.

For each of the built-in normal scopes, contexts propagate across any Java method call, including invocation of EJB local business methods. The built-in contexts do not propagate across remote method invocations or to asynchronous processes such as JMS message listeners or EJB timer service timeouts.

6.7.1. Request context lifecycle

The *request context* is provided by a built-in context object for the built-in scope type `@javax.context.RequestScoped`.

- The request scope is active during the `service()` method of any servlet in the web application and during the `doFilter()` method of any servlet filter. The request context is destroyed at the end of the servlet request, after the `service()` method and all `doFilter()` methods return.

- The request scope is active during any Java EE web service invocation. The request context is destroyed after the web service invocation completes.
- The request scope is active during any asynchronous observer method notification. The request context is destroyed after the notification completes.
- The request scope is active 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. The request context is destroyed after the remote method invocation, asynchronous method invocation, timeout or message delivery completes.

6.7.2. Session context lifecycle

The *session context* is provided by a built-in context object for the built-in passivating scope type `@javax.context.SessionScoped`.

The session scope is active during the `service()` method of any servlet in the web application and during the `doFilter()` method of any servlet filter.

The session context is shared between all servlet requests that occur in the same HTTP servlet session. The session context is destroyed when the `HTTPSession` is invalidated or times out.

6.7.3. Application context lifecycle

The *application context* is provided by a built-in context object for the built-in scope type `@javax.context.ApplicationScoped`.

- The application scope is active during the `service()` method of any servlet in the web application and during the `doFilter()` method of any servlet filter.
- The application scope is active during any Java EE web service invocation.
- The application scope is active during any asynchronous observer method notification.
- The application scope is also active 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.

The application context is shared between all servlet requests, asynchronous observer method notifications, 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 undeployed.

6.7.4. Conversation context lifecycle

The *conversation context* is provided by a built-in context object for the built-in passivating scope type `@javax.context.ConversationScoped`.

- For a JSF faces request, the context is active from the beginning of the apply request values phase, until the response is complete.
- For a JSF non-faces request, the context is active during the render response phase.

The conversation context provides access to state associated with a particular *conversation*. Every JSF request has an associated conversation. This association is managed automatically by the container according to the following rules:

- Any JSF request has exactly one associated conversation
- The conversation associated with a JSF request is determined at the end of the restore view phase and does not change during the request

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.

The container provides a built-in bean with bean type `javax.context.Conversation`, scope `@RequestScoped`, deployment type `@Standard` and binding `@Current`, named `javax.context.conversation`.

```
public interface Conversation {
   public void begin();
   public void begin(String id);
   public void end();
   public boolean isLongRunning();
   public String getId();
   public long getTimeout();
   public void setTimeout(long milliseconds);
}
```

If the conversation associated with the current JSF request is in the *transient* state at the end of a JSF request, it is destroyed, and the conversation context is also destroyed.

If the conversation associated with the current JSF request is in the *long-running* state at the end of a JSF request, it is not destroyed. Instead, it may be propagated to other requests 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 (via a navigation rule) 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 GET request parameter named `cid` containing the unique identifier of the conversation.
- The long-running conversation associated with a request may be propagated to any non-faces request via use of a GET request parameter named `cid` containing the unique identifier of the conversation. In this case, the application must manage this request parameter.

When no conversation is propagated to a JSF request, 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.
- The container is permitted to arbitrarily destroy any long-running conversation that is associated with no current JSF request, in order to conserve resources.

If the propagated conversation cannot be restored, the request is associated with a new transient conversation.

The method `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.

Open issue: allow the request to be blocked if the conversation cannot be restored.

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.

Open issue: define a mechanism for "blocking" requests. For example, allow the request to be redirected.

Chapter 7. Bean lifecycle

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 9.1, “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`.

Therefore, the actual mechanics of bean creation and destruction varies according to what kind of bean is being created or destroyed. For example:

- To create a contextual instance of a session bean, the container creates an EJB local object reference
- To create a contextual instance of a producer method bean, the container calls the producer method
- To create a contextual instance of a producer field bean, the container retrieves the current value of the field
- To create a contextual instance of a managed bean, the container calls the bean constructor
- To destroy a contextual instance of a stateful session bean, the container removes the EJB instance
- To destroy a contextual instance of a producer method bean, the container calls the disposal method, if any

7.1. Lifecycle of managed beans

Note: this lifecycle will be defined by the Managed Beans specification.

When the `create()` method of the `Bean` object that represents a managed bean is called:

- First, the container calls the bean constructor to obtain an instance of the bean, as defined in Section 5.4.3, “Injection using the bean constructor”. The container is permitted to return an instance of a container-generated subclass of the bean class, allowing interceptor and decorator bindings.
- Next, the container performs Java EE component environment injection, as required by the managed bean specification.
- Next, the container performs dependency injection, as defined in Section 5.4.4, “Injection of fields and initializer methods”.
- Finally, the container calls the `@PostConstruct` method, if any.

When the `destroy()` method is called:

- The container calls the `@PreDestroy` method, if any.
- Finally, the container destroys dependent objects, as defined in Section 5.4.5, “Destruction of dependent objects”.

7.2. 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 local reference to the session bean. This local reference behaves exactly like an ordinary EJB local object reference, except that it implements all local interfaces of the EJB. When the EJB is invoked via this local reference, the return value of `SessionContext.getInvokedBusinessInterface()` is specific to the container implementation.

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 5.4, “Dependency injection”

7.3. 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 local reference to the session bean. This local reference behaves exactly like an ordinary EJB local object reference, except that it implements all local interfaces of the EJB. When the EJB is invoked via this local reference, the return value of `SessionContext.getInvokedBusinessInterface()` is specific to the container implementation.

When the `destroy()` method is called, the container simply discards this local reference.

Note that the container performs additional work when the underlying EJB is created and removed, as defined in Section 5.4, “Dependency injection”

7.4. Lifecycle of producer methods

Any Java object may be returned by a producer method. It is not required that the returned object be an instance of another bean. However, if the returned object is not an instance of another bean, the container will provide none of the following capabilities:

- injection of other beans
- lifecycle callbacks
- method and lifecycle interception

In the following example, the producer method returns instances of other beans:

```
@SessionScoped
public class PaymentStrategyProducer {

   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, the object returned by the producer method has already had its dependencies injected, receives lifecycle callbacks and has interception enabled.

But in this example, the returned objects are not contextual instances:

```
@SessionScoped
public class PaymentStrategyProducer {

   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, the object returned by the producer method will not have any dependencies injected by the container, receives no lifecycle callbacks and does not have interception enabled.

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 5.4.6, “Invocation of producer or disposal 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 disposal method for this producer method, the container must invoke the disposal method as defined by Section 5.4.6, “Invocation of producer or disposal methods”, passing the instance given to `destroy()` to the disposed parameter.

Finally, the container destroys dependent objects.

7.5. Lifecycle of producer fields

Any Java object may be the value of a producer field. It is not required that the returned object be an instance of another bean. However, if the object is not an instance of another bean, the container will provide none of the following capabilities:

- injection of other beans
- lifecycle callbacks
- method and lifecycle interception

In the following example, the producer field contains an instance of another bean:

```
@SessionScoped
public class PaymentStrategyProducer {

   @Produces PaymentStrategy paymentStrategy;

   @CreditCard PaymentStrategy creditCard;
   @Cheque PaymentStrategy cheque;
   @Online PaymentStrategy online;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: paymentStrategy = creditCard;
         case CHEQUE: paymentStrategy = cheque;
         case ONLINE: paymentStrategy = online;
         default: throw new IllegalArgmentException();
      }
   }

}
```

In this case, the object contained by the producer field has already had its dependencies injected, received lifecycle callbacks and has interception enabled.

But in this example, the returned objects are not contextual instances:

```
@SessionScoped
public class PaymentStrategyProducer {

   @Produces PaymentStrategy paymentStrategy;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: paymentStrategy = new CreditCardPaymentStrategy();
         case CHEQUE: paymentStrategy = new ChequePaymentStrategy();
         case ONLINE: paymentStrategy = new OnlinePaymentStrategy();
```

```
         default: throw new IllegalArgmentException();
      }
   }

}
```

In this case, the object contained by the producer field does not have any dependencies injected by the container, receives no lifecycle callbacks and does not have interception enabled.

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 5.4.7, “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`.

7.6. Lifecycle of resources

An instance of a resource is a *proxy object*, provided by the container, that implements the declared bean type, delegating the actual implementation of the methods directly to the underlying Java EE component environment resource, entity manager, entity manager factory, EJB remote object or web service reference.

A resource proxy object is a dependent object of the object it is injected into.

Resource proxy objects are serializable.

The actual lifecycle of the underlying resource, entity manager, entity manager factory, remote EJB or web service is the same as for Java EE component environment injection, as defined by the Java EE platform specification.

When the `create()` method of a `Bean` object that represents a JMS resource is called, the container creates and returns a proxy object that implements the bean type of the resource.

The methods of this proxy object delegate to the underlying implementation, which is obtained using the metadata provided in the resource declaration.

- A Java EE resource is obtained using the JNDI name or mapped name specified by `@Resource`.
- A persistence context is obtained using the persistence unit name specified by `@PersistenceContext`.
- A persistence unit is obtained using the persistence unit name specified by `@PersistenceUnit`.
- A remote EJB is obtained using the JNDI name, mapped name or EJB link specified by `@EJB`.
- A web service is obtained using the JNDI name or mapped name specified by `@WebServiceRef`.

When the `destroy()` method is called, the container discards the proxy object.

7.7. Lifecycle of JMS resources

An instance of a JMS resource is a *proxy object*, provided by the container, that implements all the bean types defined in Section 3.7, “JMS resources”, delegating the actual implementation of these methods directly to the underlying JMS objects.

A JMS resource proxy object is a dependent object of the object it is injected into.

JMS resource proxy objects are serializable.

When the `create()` method of a `Bean` object that represents a JMS resource is called, the container creates and returns a proxy object that implements all the bean types of the JMS resource.

The methods of this proxy object delegate to JMS objects obtained as needed using the metadata provided by the JMS resource declaration and using standard JMS APIs.

- The `Destination` is obtained using the JNDI name or mapped name specified by `@Resource`.
- The appropriate `ConnectionFactory` for the topic or queue is obtained automatically.
- The `Connection` is obtained by calling `QueueConnectionFactory.createQueueConnection()` or `TopicConnectionFactory.createTopicConnection()`. The container is permitted to share a connection between multiple proxy objects.
- The `Session` object is obtained by calling `QueueConnection.createQueueSession()` or `TopicConnection.createTopicSession()`.
- The `MessageProducer` object is obtained by calling `QueueSession.createSender()` or `TopicSession.createPublisher()`.
- The `MessageConsumer` object is obtained by calling `QueueSession.createReceiver()` or `TopicSession.createSubscriber()`.

Open issue: or should the `ConnectionFactory` be specified using a different annotation?

When the `destroy()` method is called, the container must ensure that all JMS objects created by the proxy object are destroyed by calling `close()` if necessary.

- The `Connection` is destroyed by calling `Connection.close()` if necessary. If the connection is being shared between multiple proxy objects, the container is not required to close the connection when the proxy is destroyed.
- The `Session` object is destroyed by calling `Session.close()`.
- The `MessageProducer` object is destroyed by calling `MessageProducer.close()`.
- The `MessageConsumer` object is destroyed by calling `MessageConsumer.close()`.

The `close()` method of a JMS resource proxy object always throws an `UnsupportedOperationException`.

Chapter 8. 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 two beans.

An event comprises:

- A Java object—the *event object*
- A (possibly empty) set of instances of binding types—the *event bindings*

The event object acts as a payload, to propagate state from producer to consumer. The event bindings act as topic selectors, allowing the consumer to narrow to set of events it observes.

An event consumer observes events of a specific type, the *observed event type*, with a specific set of instances of event binding types, the *observed event bindings*.

8.1. Event types and binding types

An event object is an instance of a concrete Java class with no type variables or wildcards. The *event types* of the event include all superclasses and interfaces of the class of the event object.

An event binding type is just an ordinary binding type as specified in Section 2.3.2, “Defining new binding types” with the exception that it may be declared `@Target({FIELD, PARAMETER})`.

More formally, an event binding type is a Java annotation defined as `@Target({FIELD, PARAMETER})` or `@Target({METHOD, FIELD, PARAMETER, TYPE})` and `@Retention(RUNTIME)`. All event binding types must specify the `@javax.inject.BindingType` meta-annotation.

An event consumer will be notified of an event if the observed event type it specifies is one of the event types of the event, and if all the observed event bindings it specifies are event bindings of the event.

Every event has the binding `@javax.inject.Any`, even if it does not explicitly declare this binding.

8.2. Observer resolution algorithm

The process of matching an event to its observers is called *observer resolution*. The container considers event type and bindings when resolving observers.

Observer resolution usually occurs at runtime.

When searching for observers for an event, the container searches for observers which satisfy the following rules:

- the event object must be assignable to the observed event type, taking type parameters into consideration, and
- for each observed event binding, the event bindings must contain a matching binding with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding`.

8.2.1. Event binding types with members

As usual, the binding type may have annotation members:

```
@EventBindingType
@Target(PARAMETER)
@Retention(RUNTIME)
public @interface Role {
    String value();
}
```

Consider the following event:

```
public void login() {
```

```
    final User user = ...;
    manager.fireEvent( new LoggedInEvent(user),
            new RoleBinding() { public String value() { return user.getRole(); } );
}
```

Where `RoleBinding` is an implementation of the binding type `Role`:

```
public abstract class RoleBinding
        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 binding type member values.

8.2.2. Multiple event bindings

An event parameter may have multiple bindings:

```
public void afterDocumentUpdatedByAdmin(@Observes @Updated @ByAdmin Document doc) { ... }
```

Then this observer method will only be notified if all the observed event bindings are specified when the event is fired:

```
manager.fireEvent( document, new UpdatedBinding() {}, new ByAdminBinding() {} );
```

Other, less specific, observers will also be notified of this event:

```
public void afterDocumentUpdated(@Observes @Updated Document doc) { ... }
```

```
public void afterDocumentEvent(@Observes Document doc) { ... }
```

8.3. The `Observer` interface

An *observer* consumes events and allows the application to react to events that occur.

Observers of events implement the `javax.event.Observer` interface.

```
public interface Observer<T> {

    public void notify(T event);

}
```

8.4. Observer notification

When an event is fired by the application, the container must:

- determine the observers for that event according to the observer resolution algorithm, then,
- for each observer, call the `notify()` method of the `Observer` interface, passing the event object.

The order in which observers are called in not defined, and so portable applications should not rely upon the order in which observers are called.

Observers may throw exceptions. If an observer throws an exception, the exception aborts processing of the event. No other observers of that event will be called. The `fireEvent()` method rethrows the exception.

Any observer called before completion of a transaction may call `setRollbackOnly()` to force a transaction rollback. An observer may not directly initiate, commit or rollback JTA transactions.

8.5. Firing events

Beans fire events via an instance of the `javax.event.Event` interface, which may be injected:

```
@Any Event<LoggedInEvent> loggedInEvent;
```

The method `fire()` accepts an event object:

```
public void login() {
    ...
    loggedInEvent.fire( new LoggedInEvent(user) );

}
```

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

```
@Admin Event<LoggedInEvent> adminLoggedInEvent;
```

Or, the application may specify bindings dynamically:

```
@Any Event<LoggedInEvent> loggedInEvent;
...
LoggedInEvent event = new LoggedInEvent(user);
if ( user.isAdmin() ) {
    loggedInEvent.select( new AdminBinding() ).fire(event);
}
else {
    loggedInEvent.fire(event);
}
```

In this example, the event sometimes has the binding `@Admin`, depending upon the value of `user.isAdmin()`.

The `observe()` method registers an observer:

```
loggedInEvent.observe( new Observer<LoggedInEvent>() { public void notify(LoggedInEvent user) { ... } } );
```

8.5.1. The `Event` interface

The `Event` interface provides a method for firing events of a specific type, and a method for registering observers for events of the same type:

```
public interface Event<T> {

    public void fire(T event);
    public void observe(Observer<T> observer);

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

}
```

The `select()` method of the provided implementation of `Event` returns a child `Event` for a subtype of the event type and additional event bindings. If no subtype is given, the event type is the same as the parent.

If two instances of the same binding type are passed to `select()`, a `DuplicateBindingTypeException` is thrown.

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

The `fire()` method of the provided implementation of `Event` must call `BeanManager.fireEvent()`, passing the event type and bindings specified at the injection point. A child `Event` passes the event subtype and additional bindings, along with the bindings of its parent.

The `observe()` method of the provided implementation of `Event` must call `BeanManager.addObserver()`, passing the giv-

en observer object along with the event type and bindings specified at the injection point. A child `Event` passes the event subtype and additional bindings, along with the bindings of its parent.

8.5.2. The built-in `Instance`

The container must provide a built-in bean with:

- `Event<X>` for every legal observed event type `X` in its set of bean types,
- every event binding type in its set of binding types,
- deployment type `@Standard`,
- scope `@Dependent`,
- no bean EL name, and
- an implementation provided automatically by the container.

8.6. Observer methods

An *observer method* is an observer defined via annotations, instead of by explicitly implementing the `Observer` interface.

Unlike regular observers, observer methods are automatically discovered and registered by the container.

An observer method must be a method of a managed bean class or session bean class. An observer method may be either static or non-static. If the bean is a session bean, the observer method must be 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 bindings.

A bean may declare multiple observer methods.

8.6.1. 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 bindings of the event parameter.

If the event parameter does not explicitly declare any binding, the observer method observes events with no binding.

If the type of the event parameter contains type variables or wildcards, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

8.6.2. Declaring an observer method

A observer method may be declared by annotating a parameter `@javax.event.Observes`. That parameter is the event parameter.

```
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, as defined in Section 10.3, “Problems detected automatically by the container”.

If an observer method is annotated `@Produces`, or `@Initializer` or has a parameter annotated `@Disposes`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

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, as defined in Section 10.3, “Problems detected automatically by the container”.

The event parameter may declare bindings:

```
public void afterLogin(@Observes @Admin LoggedInEvent event) { ... }
```

8.6.3. Observer method parameters

In addition to the event parameter, observer methods may declare additional parameters, which may declare bindings. These additional parameters are injection points.

```
public void afterLogin(@Observes LoggedInEvent event, @Manager User user, @Logger Log log) { ... }
```

```
public void afterAdminLogin(@Observes @Admin LoggedInEvent event, @Logger Log log) { ... }
```

8.6.4. Conditional observer methods

Conditional observer methods are observer methods which are 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 methods may be declared by annotating the event parameter with the `@javax.event.IfExists` annotation.

```
public void refreshOnDocumentUpdate(@IfExists @Observes @Updated Document doc) { ... }
```

8.6.5. 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.

A transactional observer method may be declared by annotating the event parameter of the observer method.

```
void onDocumentUpdate(@Observes @AfterTransactionSuccess @Updated Document doc) { ... }
```

- The `@javax.event.BeforeTransactionCompletion` annotation or `<BeforeTransactionCompletion>` element specifies that the observer method is a before completion observer method.
- The `@javax.event.AfterTransactionCompletion` annotation or `<AfterTransactionCompletion>` element specifies that the observer method is an after completion observer method.
- The `@javax.event.AfterTransactionSuccess` annotation or `<AfterTransactionSuccess>` element specifies that the observer method is an after success observer method.
- The `@javax.event.AfterTransactionFailure` annotation or `<AfterTransactionFailure>` element specifies that the observer method is an after failure observer method.

A transactional observer method may not specify more than one of the four types. If a transactional observer method specifies more than one of the four types, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

8.6.6. Asynchronous observer methods

Asynchronous observer methods are observer methods which receive event notifications asynchronously.

An asynchronous observer method may be declared by annotating the event parameter of the observer method `@javax.event.Asynchronously`.

```
void onDocumentUpdate(@Observes @Asynchronously @Updated Document doc) { ... }
```

An asynchronous observer method may also be a transactional observer method. However, it may not be a before completion observer method or a conditional observer method. If an asynchronous observer method is specified as a before completion or conditional observer method, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

8.6.7. Observer object for an observer method

For every observer method of an enabled bean, the container is responsible for providing and registering an appropriate implementation of the `Observer` interface, that delegates event notifications to the observer method, by calling the observer method as defined in Section 5.4.8, “Invocation of observer methods”.

The `notify()` method of the `Observer` implementation for an observer method either invokes the observer method immediately, or asynchronously, or registers the observer method for later invocation during the transaction completion phase, using a JTA `Synchronization`.

- If the observer method is an asynchronous transactional observer method and there is currently a JTA transaction in progress, the observer object calls the observer method asynchronously during the after transaction completion phase.
- Otherwise, if the observer method is a transactional observer method and there is currently a JTA transaction in progress, the observer object calls the observer method during the appropriate transaction completion phase.
- Otherwise, if the observer method is an asynchronous observer method, the container calls the observer method asynchronously.
- Otherwise, the container calls the observer immediately.

The container is not required to guarantee delivery of asynchronous events in the case of a server shutdown or failure.

Observer methods may throw exceptions:

- If the observer is a transactional or asynchronous observer method, any exception is caught and logged by the container.
- Otherwise, the exception is rethrown by the `notify()` method of the observer object. If the exception is a checked exception, it is wrapped and rethrown as an (unchecked) `ObserverException`.

8.6.8. Observer 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 an asynchronous observer method, it is called with no active transaction, no client security context and with a new request context that is destroyed when the observer method returns. The application context is also active.
- Otherwise, if the observer method is a `@BeforeTransactionCompletion` 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()`.

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.

8.7. JMS event mappings

An event type may be mapped to JMS topic.

An *event mapping* is a special kind of observer method that is declared by an interface, for example:

```
interface EventMappings {
    void mapLoggedInEvent(@Observes LoggedInEvent event, @Events Topic eventTopic);
}
```

Where the parameter of type `Topic` resolves to the following JMS resource:

```
@Resource(name="java:global/env/jms/Events")
@Produces @Events Topic eventTopic;
```

The event parameter specifies the *mapped event type and bindings*. Every JMS resource representing a topic that any injected parameter resolves to is a *mapped topic*.

An event mapping may be specified as a member of any interface.

All observers of mapped event types must be asynchronous observer methods. If an observer for a mapped event type is not an asynchronous observer method, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

For every event mapping, the container must:

- send a message containing the serialized event and its event bindings to every mapped topic whenever an event with the mapped event type and bindings is fired, and
- monitor every mapped topic for messages containing events of that mapped event type and bindings and notify all local observers whenever a message containing an event is received.

Thus, events with the mapped event type and bindings are distributed to other processes which have the same event mapping.

Chapter 9. Framework integration and the bean manager

A third-party framework way 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

Bean definitions provided by a third-party framework may be associated with a certain *activity*.

Third-party framework integration is enabled via the important SPI interfaces `Bean` and `BeanManager`.

9.1. The `Bean` interface

The interface `javax.inject.spi.Bean` defines everything the container needs to manage instances of a certain bean.

```
public interface Bean<T>
        extends Contextual<T> {

    public Set<Type> getTypes();
    public Set<Annotation> getBindings();
    public Class<? extends Annotation> getScopeType();
    public Class<? extends Annotation> getDeploymentType();
    public String getName();

    public boolean isSerializable();
    public boolean isNullable();

    public Set<InjectionPoint> getInjectionPoints();

}
```

Note that implementations of `Bean` must implement the operations defined by the `Contextual` interface defined in Section 6.1, “The Contextual interface”.

An instance of `Bean` exists for every enabled bean in a deployment.

An application or third party framework may add support for new kinds of beans beyond those defined by the this specification (managed beans, session beans, producer methods and fields, resources and JMS resources) by implementing `Bean` and registering beans with the container, using the mechanism defined in Section 9.2.6, “Bean registration”.

9.2. The `BeanManager` object

Third-party frameworks sometimes interact directly with the container via programmatic API call. The interface `javax.inject.spi.BeanManager` provides operations for obtaining contextual references for beans, along with many other operations of use to third-party frameworks.

The container provides a built-in bean with bean type `BeanManager`, scope `@Dependent`, deployment type `@Standard` and binding `@Current`. Thus, any bean may obtain an instance of `BeanManager` by injecting it:

```
@Current BeanManager manager;
```

Alternatively, a framework may obtain the `BeanManager` object from JNDI. The container must register an instance of `BeanManager` with name `java:app/BeanManager` in JNDI at deployment time.

Open issue: should it go in java:app or java:comp or both?

Any operation of `BeanManager` may be called at any time during the execution of the application.

9.2.1. Obtaining a contextual reference for a bean

The method `BeanManager.getReference()` returns a contextual reference for a given bean, as defined in Section 6.5.3,

"Contextual reference for a bean".

```
public interface BeanManager {

    public <T> T getReference(Bean<T> bean);

    ...

}
```

9.2.2. Obtaining an injectable reference

The method `BeanManager.getInjectableReference()` returns an injectable reference for a given injection point, as defined in Section 5.4.1, "Injectable references".

```
public interface BeanManager {

    public <T> T getInjectableReference(InjectionPoint ij, CreationalContext<?> ctx);

    ...

}
```

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()`.

9.2.3. Obtaining a `Bean` by type

The `getBeans()` method of the `BeanManager` interface returns the result of the typesafe resolution algorithm defined in Section 5.1, "Typesafe resolution algorithm".

```
public interface BeanManager {

    public <T> Set<Bean<T>> getBeans(Class<T> apiType, Annotation... bindings);
    public <T> Set<Bean<T>> getBeans(TypeLiteral<T> apiType, Annotation... bindings);
    public Set<Bean<?>> getBeans(Type apiType, Annotation... bindings);

    ...

}
```

If no bindings are passed to `getBeans()`, the default binding `@Current` is assumed.

If a parameterized type with a type parameter or wildcard is passed to `getBeans()`, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `getBeans()`, a `DuplicateBindingTypeException` is thrown.

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

9.2.4. Obtaining a `Bean` by name

The `getBeans()` method of the `BeanManager` interface returns the result of the name resolution algorithm defined in Section 5.2, "Name resolution algorithm".

```
public interface BeanManager {

    public Set<Bean<?>> getBeans(String name);

    ...

}
```

9.2.5. Obtaining the most specialized bean

The method `BeanManager.getMostSpecializedBean()` returns the `Bean` object representing the most specialized enabled bean registered with the container that specializes the given bean, as defined in Section 4.3.2, “Direct and indirect specialization”.

```
public interface BeanManager {

    public <X> Bean<X> getMostSpecializedBean(Bean<X>);

    ...

}
```

9.2.6. `Bean` registration

The `BeanManager.addBean()` method registers a new bean with the container, thereby making it available for injection into other beans.

```
public interface BeanManager {

    public BeanManager addBean(Bean<?> bean);

    ...

}
```

The `Bean` parameter may represent an interceptor or decorator.

9.2.7. Observer registration

An observer instance may be registered with the container by calling `BeanManager.addObserver()`:

```
public interface BeanManager {

    public <T> BeanManager addObserver(Observer<T> observer, Class<T> eventType,
                                       Annotation... bindings);
    public <T> BeanManager addObserver(Observer<T> observer, TypeLiteral<T> eventType,
                                       Annotation... bindings);
    public BeanManager addObserver(Observer<?> observer, Type eventType,
                                   Annotation... bindings);

    ...

}
```

The first parameter is the observer object. The second parameter is the observed event type. The remaining parameters are optional observed event bindings. The observer is notified when an event object that is assignable to the observed event type is raised with the observed event bindings.

An observer instance may be deregistered by calling `BeanManager.removeObserver()`:

```
public interface BeanManager {

    public <T> BeanManager removeObserver(Observer<T> observer, Class<T> eventType,
                                          Annotation... bindings);
    public <T> BeanManager removeObserver(Observer<T> observer, TypeLiteral<T> eventType,
                                          Annotation... bindings);
    public BeanManager removeObserver(Observer<?> observer, Type eventType,
                                      Annotation... bindings);

    ...

}
```

If the observed event type passed to `addObserver()` or `removeObserver()` contains type variables or wildcards, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `addObserver()` or `removeObserver()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `addObserver()` or `removeObserver()`, an `IllegalArgumentException` is thrown.

9.2.8. Firing an event

The method `BeanManager.fireEvent()` fires an event and notifies observers, according to Section 8.4, “Observer notification”.

```
public interface BeanManager {

    public void fireEvent(Object event, Annotation... bindings);

    ...

}
```

The first argument is the event object. The remaining parameters are event bindings.

If the type of the event object passed to `fireEvent()` contains type variables or wildcards, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `fireEvent()`, a `DuplicateBindingTypeException` is thrown.

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

9.2.9. Observer resolution

The method `BeanManager.resolveObservers()` resolves observers for an event according to the observer resolution algorithm defined in Section 8.2, “Observer resolution algorithm”.

```
public interface BeanManager {

      public <T> Set<Observer<T>> resolveObservers(T event, Annotation... bindings);

      ...

}
```

The first parameter of `resolveObservers()` is the event object. The remaining parameters are event bindings.

If the type of the event object passed to `resolveObservers()` contains type variables or wildcards, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `resolveObservers()`, a `DuplicateBindingTypeException` is thrown.

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

9.2.10. Dependency validation

The `BeanManager.validate()` operation validates a dependency:

```
public interface BeanManager {

    public void validate(InjectionPoint injectionPoint);
    ...

}
```

The method `validate()` validates the dependency and throws an `IllegalStateException` if there is a deployment problem (for example, an unsatisfied or ambiguous dependency) associated with the injection point.

9.2.11. Enabled deployment types

The method `BeanManager.getEnabledDeploymentTypes()` exposes the list of enabled deployment types, in order of lower to higher precedence, as defined by Section 2.5.5, “Enabled deployment types”.

```
public interface BeanManager {
```

```
    public List<Class<?>> getEnabledDeploymentTypes();

    ...
}
```

Third-party frameworks may use this method to inspect meta-annotations that appear on the deployment types and thereby discover information about the deployment.

9.2.12. Registering a `Context`

A custom implementation of `Context` may be associated with a scope type by calling `BeanManager.addContext()`.

```
public interface BeanManager {

    public BeanManager addContext(Context context);

    ...
}
```

9.2.13. 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 interface BeanManager {

    public Context getContext(Class<? extends Annotation> scopeType);

    ...
}
```

9.3. Alternative metadata sources

A third-party framework may provide an alternative metadata source, such as configuration by XML.

The interfaces `AnnotatedType`, `AnnotatedField`, `AnnotatedMethod`, `AnnotatedConstructor` and `AnnotatedParameter` in the package `javax.inject.spi` allow a third-party framework to specify metadata that overrides the annotations that exist on a bean class. The third-party framework is responsible for implementing the interfaces, thereby exposing the metadata to the container.

```
public interface AnnotatedType<X> extends Annotated {

    public Class<X> getJavaClass();

    public Set<AnnotatedConstructor<X>> getConstructors();
    public Set<AnnotatedMethod<X>> getMethods();
    public Set<AnnotatedField<X>> getFields();

}
```

```
public interface AnnotatedField<X> extends Annotated {

    public Field getJavaMember();

    public AnnotatedType<X> getDeclaringType();

}
```

```
public interface AnnotatedCallable<X> extends Annotated {

    public Member getJavaMember();

    public AnnotatedType<X> getDeclaringType();
    public List<AnnotatedParameter<X>> getParameters();

}
```

```
public interface AnnotatedMethod<X> extends AnnotatedCallable<X> {

    public Method getJavaMember();


}
```

```
public interface AnnotatedConstructor<X> extends AnnotatedCallable<X> {

    public Constructor<X> getJavaMember();

}
```

```
public interface AnnotatedParameter<X> extends Annotated {

    public int getPosition();
    public AnnotatedCallable<X> getDeclaringCallable();

}
```

The interface `Annotated` exposes the overriding annotations and type declarations.

```
public interface Annotated {

    Type getType();

    public <T extends Annotation> T getAnnotation(Class<T> annotationType);
    public Set<Annotation> getAnnotations();
    public boolean isAnnotationPresent(Class<? extends Annotation> annotationType);

}
```

9.4. Helper objects for `Bean` implementations

Third-party frameworks sometimes provide custom implementations of `Bean` or inject dependencies directly into objects which are not contextual bean instances. To simplify the implementation of these frameworks, the `BeanManager` provides access to helper objects which parse and validate the standard metadata defined by this specification and perform injection upon an object according to the standard injection lifecycle defined in Section 5.4.3, “Injection using the bean constructor”.

```
public interface BeanManager {

    public <T> InjectionTarget<T> createInjectionTarget(Class<T> type);
    public <T> InjectionTarget<T> createInjectionTarget(AnnotatedType<T> type);

    public <T> ManagedBean<T> createManagedBean(Class<T> type);
    public <T> ManagedBean<T> createManagedBean(AnnotatedType<T> type);

    ...

}
```

The method `createInjectionTarget()` returns an instance of `InjectionTarget` representing the given type, or throws an `IllegalArgumentException` if there is a definition error associated with any injection point of the type.

The method `createManagedBean()` returns an instance of `ManagedBean` representing the given managed bean type, or throws an `IllegalArgumentException` if there is any kind of definition error associated with the type.

When an `AnnotatedType` is passed to `createInjectionTarget()` or `createManagedBean()` the container ignores the annotations and types declared by the elements of the actual Java class and uses the metadata provided via the `Annotated` interface instead.

The `InjectionTarget` interface provides operations for performing dependency injection upon instances of a type.

```
public interface InjectionTarget<X> {

    public X instantiate();
    public void inject(X instance);
    public void postConstruct(X instance);
    public void preDestroy(X instance);
    public void destroy(X instance);

```

```
    public Set<InjectionPoint> getInjectionPoints();

}
```

The method `instantiate()` calls the constructor annotated `@Initializer` if it exists, or the constructor with no parameters otherwise, as defined in Section 5.4.3, “Injection using the bean constructor”.

The method `inject()` performs dependency injection upon the given object, first setting the value all injected fields, and then calling all the initializer methods, as defined in Section 5.4.4, “Injection of fields and initializer methods”.

The methods `postConstruct()` and `preDestroy()` call the `@PostConstruct` and `@PreDestroy` callbacks respectively, if they exist, according to the semantics required by the Java EE platform specification.

The method `destroy()` destroys dependent objects of the given object, as defined in Section 5.4.5, “Destruction of dependent objects”.

The method `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all injected fields, bean constructor parameters and initializer method parameters.

The `ManagedBean` interface exposes bean-level metadata for a managed bean, operations for performing dependency injection upon instances of the bean, and operations for discovering producer methods and observer methods of the bean.

```
public interface ManagedBean<X>
        extends Bean<X>, InjectionTarget<X> {

    public Set<ProducerBean<X, ?>> getProducerBeans();
    public Set<ObserverMethod<X, ?>> getObserverMethods();

}
```

The method `getProducerBeans()` returns a set of `ProducerBean` objects representing the producer methods and fields of the bean.

The method `getObserverMethods()` returns a set of `ObserverMethod` objects representing the observer methods of the bean.

The `ProducerBean` interface exposes bean-level metadata for a producer method or field and operations for producing and disposing instances of the bean.

```
public interface ProducerBean<X, T>
        extends Bean<T> {

    public T produce(X bean);
    public void dispose(T instance);
    public void destroy(T instance);

}
```

The method `produce()` calls the producer method upon the given bean instance, as defined in Section 5.4.6, “Invocation of producer or disposal methods”.

The method `produce()` calls the disposal method upon the given bean instance, as defined in Section 5.4.6, “Invocation of producer or disposal methods”.

The method `destroy()` destroys dependent objects of the given object, as defined in Section 5.4.5, “Destruction of dependent objects”.

The `ObserverMethod` interface exposes the observed event type and observed event binding types for an observer method, an operation for calling the method, and an instance of `Observer` for the method.

```
public interface ObserverMethod<X, T> {

    void call(X instance, T event);

    public Observer<T> getObserver();

    public Type getObservedEventType();
    public Set<Annotation> getObservedEventBindings();

    public Set<InjectionPoint> getInjectedParameters();
```

```
}
```

The method `call()` calls the observer method, upon the given bean instance as defined in Section 5.4.8, “Invocation of observer methods”.

The method `getObserver()` returns the observer object for the observer method defined in Section 8.6.7, “Observer object for an observer method”.

The methods `getObservedEventType()` and `getObservedEventBindings()` return the observed event type and bindings of the observer method.

The method `getInjectedParameters()` returns the set of `InjectionPoint` objects representing all injected method parameters.

Open issue: should we provide `BeanManager.addObserverMethod()` to return an instance of `ObserverMethod` for a given method?

9.5. Activities

Bean definitions may be scoped to an *activity*. This specification only provides a programmatic API for defining activities, since this feature is intended for use with third-party orchestration frameworks that integrate with the container.

Activities are represented by instances of `BeanManager`. The method `createActivity()` creates a new child activity of an activity:

```
public interface BeanManager {

    public BeanManager createActivity();

    ...

}
```

A child activity inherits all beans, interceptors, decorators, observers, and contexts defined by its direct and indirect parent activities:

- every bean belonging to a parent activity also belongs to the child activity, is eligible for injection into other beans belonging to the child activity and may be obtained by dynamic lookup via the child activity,
- every interceptor and decorator belonging to a parent activity also belongs to the child activity and may be applied to any bean belonging to the child activity,
- every observer belonging to a parent activity also belongs to the child activity and receives events fired via the child activity, and
- every context object belonging to the parent activity also belongs to the child activity.

Beans and observers may be registered with an activity by calling `addBean()` or `addObserver()` on the `BeanManager` object that represents the activity.

Beans and observers registered with an activity are visible only to that activity and its children—they are never visible to direct or indirect parent activities, or to other children of the parent activity:

- a bean registered with the child activity is not available for injection into any bean registered with a parent activity,
- a bean registered with a child activity is not available for injection into a non-contextual instance,
- a bean registered with a child activity may not be obtained by dynamic lookup via the parent activity, and
- an observer registered with the child activity does not receive events fired via a parent activity.

If a bean registered with a child activity has the bean type and all bindings of some injection point of some bean registered with a direct or indirect parent activity, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

Interceptors and decorators may not be registered with a child activity. The `addInterceptor()` and `addDecorator()` methods throw `UnsupportedOperationException` when called on a `BeanManager` object that represents a child activity.

9.5.1. Current activity

An activity may be associated with the current context for a normal scope by calling `setCurrent()`, passing the normal scope type:

```
public interface BeanManager {

    public BeanManager setCurrent(Class<? extends Annotation> scopeType);

    ...

}
```

If the given scope is inactive when `setCurrent()` is called, a `ContextNotActiveException` is thrown. If the given scope type is not a normal scope, an `IllegalArgumentException` is thrown.

All EL evaluations (as defined Section 5.6, “Integration with Unified EL”), all calls to any injected `BeanManager` object or `BeanManager` object obtained via JNDI lookup (as defined by Section 9.2, “The BeanManager object”), all calls to any injected `Event` object (as defined in Section 8.5.1, “The Event interface”) and all calls to any injected `Instance` object (as defined by Section 5.5.1, “The Instance interface”) are processed by the *current activity*:

- If the root activity has no active normal scope such that the current context for that scope has an associated activity, the root activity is the current activity.
- If the root activity has exactly one active normal scope such that the current context for that scope has an associated activity, that activity is the current activity.
- Otherwise, there is no well-defined current activity, and the behavior is undefined. Portable frameworks and applications should not depend upon the behavior of the container when two different current contexts have an associated activity.

A bean registered with an activity is only available to Unified EL expressions that are evaluated when that activity or one of its children is the current activity.

Chapter 10. Packaging and deployment

When an application is deployed, the container must perform *bean discovery*, detect definition errors and deployment problems and raise events that allow third-party frameworks to integrate with the deployment lifecycle.

Bean discovery is the process of determining:

- What beans, interceptors and decorators *exist* in the deployment archive
- Which beans, interceptors and decorators are *enabled* for this deployment
- The *precedence* of the enabled beans, and the *ordering* of enabled interceptors and decorators

Bean classes must be deployed in an EAR, JAR, WAR, EJB-JAR or RAR archive or directory in the application classpath that has a file named `beans.xml` in the metadata directory (`META-INF`, or `WEB-INF` in the case of a WAR). If a bean is deployed to a location that is not in the application classpath, or does not contain a file named `beans.xml` in the metadata directory, it will not be discovered by the container.

Additional beans may be registered programatically with the container by the application or third-party framework after the automatic bean discovery completes. Third-party frameworks may even provide the ability to register certain bean definitions with a *activity*, thereby limiting their visibility to certain contexts.

10.1. Deployment lifecycle

When an application is deployed, the container performs the following steps:

- First, the container searches for static observer methods of type `BeanDefinition` and fires an event of that type.
- Next, the container performs bean discovery and registers `Bean` and `Observer` objects for the discovered beans. The container detects definition errors by validating the bean classes and metadata and aborts deployment of the application if any definition errors exist, as defined in Section 10.3, “Problems detected automatically by the container”.
- Next, the container fires an event of type `AfterBeanDiscovery`, allowing the application or third-party frameworks to register additional `Bean` and `Observer` objects.
- Next, the container detects deployment problems by validating bean dependencies and specialization and aborts deployment of the application if any deployment problems exist, as defined in Section 10.3, “Problems detected automatically by the container”.
- Next, the container fires an event of type `AfterDeploymentValidation`.
- Finally, the container begins directing requests to the application.

10.2. Bean discovery

When bean discovery occurs, the container considers:

- any `beans.xml` file in any metadata directory of the application classpath,
- any `ejb-jar.xml` file in any metadata directory of the application classpath that also contains a `beans.xml` file, and
- any Java class in any archive or directory in the classpath that has a `beans.xml` file in the metadata directory.

The container automatically discovers managed beans (according to the rules of Section 3.2.1, “Which Java classes are managed beans?”) and session beans deployed and/or declared in these locations and searches the bean classes for producer methods, producer fields, disposal methods and observer methods.

Next, the container determines which beans, interceptors and decorators are enabled, according to the rules defined in Section 2.5.5, “Enabled deployment types”, Section A.3.8, “Interceptor enablement and ordering” and Section A.5.4, “Decorator enablement and ordering”, taking into account any `<Deploy>`, `<Interceptors>` and `<Decorators>` declarations in the `beans.xml` files.

Next, the container creates and registers `Bean` objects (that implement the rules of Chapter 7, *Bean lifecycle*) and `Observer` objects.

- For each enabled bean that is not an interceptor or decorator, the container creates and registers an instance of `Bean`.
- For each enabled interceptor, the container creates and registers an instance of `Interceptor`.
- For each enabled decorator, the container creates and registers an instance of `Decorator`.
- For each observer method of an enabled bean, the container creates and registers an instance of `Observer` that implements the rules of Section 8.6.7, “Observer object for an observer method”.

10.3. 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
- Deployment problems—occur when there are problems resolving dependencies, or inconsistent specialization, in a particular deployment
- Execution errors—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 deployment time. If a definition error exists in a deployment, the deployment will be aborted by the container.

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

Execution errors may not be detected until they actually occur at runtime.

Execution errors are represented by instances of `javax.inject.InjectionException` and its subclasses.

```
public class ExecutionException extends RuntimeException {

    public ExecutionException(String message) { ... }

}
```

This specification defines the following subclasses:

- `CreationException`
- `IllegalProductException`
- `ObserverException`
- `DuplicateBindingTypeException`
- `ContextNotActiveException`
- `AmbiguousResolutionException`
- `UnsatisfiedResolutionException`

10.4. Initialization events

The container fires events, allowing third-party frameworks to integrate with the container initialization process.

10.4.1. `BeforeBeanDiscovery` event

The container must fire an event before it begins the bean discovery process. The event object must be of type `BeforeBeanDiscovery`:

```
public interface BeforeBeanDiscovery {
    BeanDiscovery declareBindingType(Class<? extends Annotation> bindingType);
    BeanDiscovery declareScopeType(Class<? extends Annotation> scopeType);
    BeanDiscovery declareStereotype(Class<? extends Annotation> stereotype);
    BeanDiscovery declareInterceptorBindingType(Class<? extends Annotation> bindingType);
}
```

The operations of the `BeforeBeanDiscovery` instance allow a third-party framework to declare that any annotation as a binding type, scope type, stereotype or interceptor binding type.

Since this event occurs before bean discovery takes place, observers of this event must be static methods.

```
static 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.

10.4.2. `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 `Observer` objects for the discovered beans, but before detecting deployment problems.

The event object must be of type `AfterBeanDiscovery`:

```
public interface AfterBeanDiscovery {
    AfterBeanDiscovery addDefinitionError(Throwable e);
    boolean hasDefinitionError();
}
```

The method `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after all observers have been notified.

Any bean may observe this event.

```
void afterBeanDiscovery(@Observes AfterBeanDiscovery event, BeanManager manager) { ... }
```

A third party framework might take advantage of this event to register beans and interceptors with the container.

If any observer method of the `AfterBeanDiscovery` event throws an exception, the exception is treated as a definition error by the container.

10.4.3. `AfterDeploymentValidation` event

The container must fire a third event after it has validated that there are no deployment problems and before the deployment begins processing requests.

The event object must be of type `AfterDeploymentValidation`:

```
public interface AfterDeploymentValidation {
    AfterDeploymentValidation addDeploymentProblem(Throwable e);
    boolean hasDeploymentProblem();
}
```

The method `addDeploymentProblem()` registers a deployment problem with the container, causing the container to abort deployment after all observers have been notified.

Any bean may observe this event.

```
void afterDeploymentValidation(@Observes AfterDeploymentValidation event, BeanManager manager) { ... }
```

If any observer method of the `AfterDeploymentValidation` event throws an exception, the exception is treated as a de-

ployment problem by the container.

The container must not allow any request to be processed by the deployment until all observers of this event return.

The request and application contexts are active when these events are fired.

Appendix A. Interceptors and decorators

The following functionality is to be integrated with the existing interceptor functionality defined by the EJB specification and removed from this specification.

Managed beans and EJB session and message-driven beans support interception as defined by the package `javax.interceptor`. Interceptors may be bound to a bean using the `javax.interceptor.Interceptors` annotation, or by using an *interceptor binding*.

Interceptors are usually used to implement *cross-cutting* concerns, functionality that is orthogonal to the type system. In addition, this specification provides support for *decorators*. A decorator intercepts method invocations for a specific bean type. Unlike interceptors, decorators are typesafe, and cannot be used to implement cross-cutting concerns.

Interceptors and decorators may be bound to any managed bean that is not itself an interceptor or decorator or to any EJB session or message-driven bean.

A.1. Business methods

Method interception by interceptors and decorators applies to *business method invocations* of a managed bean or EJB session or message driven bean.

Open issue: will the managed bean spec define the notion of a business method? What about self-invocations?

Invocations of initializer methods by the container during bean creation are not considered to be business method invocations.

Invocations of `@PreDestroy` and `@PostConstruct` callbacks by the container are not considered to be business method invocations.

All invocations of producer methods, disposal methods and observer methods *are* considered to be business method invocations.

A.2. 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 {

    @LoggedIn User user;

    @AroundInvoke public void authorize(InvocationContext ic) {
        try {
            if ( !user.isBanned() ) {
                System.out.println("Authorized");
                ic.proceed();
            }
            else {
                System.out.println("Not authorized");
                throw new NotAuthorizedException();
            }
        }
        catch (NotAuthenticatedException nae) {
            System.out.println("Not authenticated");
            throw nae;
        }
    }

}
```

The `@Interceptor` annotation identifies the `AuthorizationInterceptor` class as an interceptor. The `@Secure` annotation is a custom *interceptor binding type*.

```
@Inherited
```

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

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

```
@Model
public class DocumentEditor {

    @Current Document document;
    @LoggedIn User user;
    @PersistenceContext 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.

A.3. Interceptors

An *interceptor* is a managed bean with a bean class that is also an interceptor class as defined by the EJB specification.

An interceptor with scope `@Dependent` must be serializable. If an interceptor has scope `@Dependent` and is not serializable, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

An interceptor may be either a business method interceptor, a lifecycle callback interceptor or both.

A.3.1. Business method interceptors

An interceptor method for business method invocations is a method of an interceptor with return type `Object` and a single parameter of type `javax.interceptor.InvocationContext`, annotated `@AroundInvoke`.

Interceptor methods for business method invocations are called by the container when a business method is invoked.

If an interceptor has an interceptor method for business method invocations, we describe it as a *business method interceptor*.

A.3.2. Lifecycle callback interceptors

An interceptor method for a lifecycle callback is a method of an interceptor bean class with return type `void` and a single parameter of type `javax.interceptor.InvocationContext`, annotated `@PostConstruct`, `@PreDestroy`, `@PrePassivate` or `@PostActivate`.

Interceptor methods for a lifecycle callbacks are called by the container when the corresponding `@PostConstruct`, `@PreDestroy`, `@PrePassivate` or `@PostActivate` events occur.

If an interceptor has an interceptor method for a lifecycle callback, we describe it as a *lifecycle callback interceptor*.

A.3.3. Declaring an interceptor

An interceptor may be declared by annotating the interceptor bean class with the `@Interceptor` stereotype.

A.3.4. Support for `@Interceptors`

Any bean class may declare interceptors using `@Interceptors`. The semantics are completely defined by the EJB specification.

A.3.5. Interceptor bindings

As an extension to the functionality defined by the `javax.interceptor` package, this specification provides an alternative method of binding interceptors to managed beans and EJB session and message-driven beans.

Even when interceptors are bound using this mechanism, the interception semantics are defined by the EJB specification.

An *interceptor binding type* is a Java annotation defined as `@Target({TYPE, METHOD})` or `@Target(TYPE)` and `@Retention(RUNTIME)`.

An interceptor binding type may be declared by specifying the `@InterceptorBindingType` meta-annotation.

```
@Inherited
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transactional {}
```

Multiple interceptors may be bound to the same interceptor binding type or types.

A.3.5.1. Interceptor binding types with additional interceptor bindings

An interceptor binding type may declare other interceptor bindings.

```
@Inherited
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Transactional
public @interface DataAccess {}
```

```
<myfwk:DataAccess>
    <InterceptorBindingType/>
    <myfwk:Transactional/>
</myfwk: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})`.

A.3.5.2. Interceptor bindings for stereotypes

Interceptor bindings may be applied to a stereotype by annotating the stereotype annotation:

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

An interceptor binding declared by a stereotype are inherited by any bean that declares that stereotype.

If a stereotype declares interceptor bindings, it must be defined as `@Target(TYPE)`.

A.3.6. Declaring the interceptor bindings of an interceptor

Interceptor bindings for interceptors declared using `@Interceptor` are specified by annotation the interceptor bean class.

```
@Transactional @Interceptor
public class TransactionInterceptor {

   @AroundInvoke
   public Object manageTransaction(InvocationContext ctx) { ... }
```

```
}
```

All interceptors declared using `@Interceptor` must specify at least one interceptor binding.

If an interceptor declared using `@Interceptor` does not declare any interceptor binding, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

Lifecycle callback interceptors may only declare interceptor binding types that are defined as `@Target(TYPE)`. If a lifecycle callback interceptor declares an interceptor binding type that is defined `@Target({TYPE, METHOD})`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

A.3.7. Binding an interceptor to a bean

A lifecycle callback interceptor may be bound to a bean by declaring, at the class level, the same interceptor bindings that were declared by the interceptor.

A business method interceptor may be bound to all non-static, non-private, non-final methods of a bean by declaring the same interceptor bindings, at the class level, that were declared by the interceptor.

A business method interceptor may be bound to a non-static, non-private, non-final method of a bean by declaring the same interceptor bindings, at the method level, that were declared by the interceptor.

If a managed bean class that is not an interceptor or decorator is declared final, or has any non-static, non-private, final methods, and also declares an interceptor binding or a stereotype with interceptor bindings, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a non-static, non-private method of a managed bean class is declared final and also declares an interceptor binding, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

A.3.7.1. Binding an interceptor

Interceptor bindings may be declared by annotating the bean class with an 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() { ... }

}
```

Interceptors may be enabled or disabled at deployment time. Disabled interceptors are never called at runtime.

A.3.8. Interceptor enablement and ordering

By default, interceptors bound via interceptor bindings are not enabled. An interceptor must be explicitly enabled by listing its bean class under the `<Interceptors>` element in `beans.xml`.

```
<Interceptors>
    <myfwk:TransactionInterceptor/>
    <myfwk:LoggingInterceptor/>
</Interceptors>
```

The order of the interceptor declarations determines the interceptor ordering. Interceptors which occur earlier in the list are called first.

If a class listed under the `<Interceptors>` element is not the bean class of at least one interceptor, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

If the bean class of an interceptor with a disabled deployment type is listed under the `<Interceptors>` element, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

If the `<Interceptors>` element is specified in more than one `beans.xml` document, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

Interceptors declared using `@Interceptors` or in `ejb-jar.xml` are called before interceptors declared using interceptor bindings.

Interceptors are called before decorators.

A.3.9. The `Interceptor` object for an interceptor

The `Bean` object for an interceptor must implement `Interceptor`.

```
public interface Interceptor extends Bean<Object> {

    public Set<Annotation> getInterceptorBindingTypes();

    public Method getMethod(InterceptionType type);

}
```

An `InterceptionType` identifies the kind of lifecycle callback or business method.

```
public enum InterceptionType {
    AROUND_INVOKE, POST_CONSTRUCT, PRE_DESTROY, PRE_PASSIVATE, POST_ACTIVATE
}
```

The `getMethod()` method returns the interceptor method for the specified kind of lifecycle callback or business method. The `getMethod()` method must return a null value if the interceptor does not intercepts callbacks or business methods of the given type.

A.3.10. Interceptor resolution

The following method returns the ordered list of enabled interceptors for a set of interceptor bindings.

```
public interface BeanManager {

    List<Interceptor> resolveInterceptors(InterceptionType type,
                                          Annotation... interceptorBindings);

    ...

}
```

If two instances of the same interceptor binding type are passed to `resolveInterceptors()`, a `DuplicateBindingTypeException` is thrown.

If no interceptor binding type instance is passed to `resolveInterceptors()`, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not an interceptor binding type is passed to `resolveInterceptors()`, an `IllegalArgumentException` is thrown.

The following algorithm must be used by the container when resolving interceptors:

- First, the container identifies the set of *matching* enabled interceptors where for each declared interceptor binding,

there exists an interceptor binding in the set of given bindings or, recursively, meta-annotations of those binding types, with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding` (see Section A.3.10.2, “Interceptor binding types with members”).

- Next, the container narrows the set of matching interceptors according to whether the interceptor intercepts the given kind of lifecycle callback or business method.
- Next, the container orders the matching interceptors according to the interceptor ordering specified in Section A.3.8, “Interceptor enablement and ordering” and returns the resulting list of interceptors. If no matching interceptors exist in the set, an empty list is returned.

A.3.10.1. Interceptors with multiple bindings

An interceptor class may specify multiple interceptor bindings, in which case the interceptor will be applied only to beans that declare all the bindings at the class level, and to methods of beans for which every binding appears at either the method or class level.

Consider the following interceptor:

```
@Transactional @Secure @Interceptor
public class TransactionalSecurityInterceptor {

   @AroundInvoke
   public void aroundInvoke() { ... }

}
```

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() { ... }

}
```

A.3.10.2. Interceptor binding types with members

According to the interceptor resolution algorithm defined above, interceptor binding types may have annotation members.

This interceptor binding type declares a member:

```
@Inherited
@InterceptorBindingType
@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) { ... }

}
```

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
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transactional {
   @NonBinding boolean requiresNew() default false;
}
```

Array-valued or annotation-valued members of an interceptor binding type must be annotated `@NonBinding`. If an array-valued or annotation-valued member of an interceptor binding type is not annotated `@NonBinding`, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

A.3.11. Interceptor stack creation

When a bean with interceptors is created, the container must:

- Identify the interceptors for each lifecycle callback and business method by calling `BeanManager.resolveInterceptors()` passing the interceptor bindings for the callback or business method, including all interceptor bindings defined at the class level, method level and by stereotypes.
- Identify the interceptors defined using the `@Interceptors` annotation for each lifecycle callback and business method.
- For each unique interceptor, call `BeanManager.getReference()`, passing the `Interceptor` object, to obtain an instance of the interceptor. For a given interceptor and a given intercepted instance, the container must call `BeanManager.getReference()` exactly once.
- For each lifecycle callback and business method build an ordered list of returned interceptor instances.

The resulting ordered lists of interceptor instances are called *interceptor stacks*.

A.3.12. Interceptor invocation

Whenever a business method or lifecycle callback is invoked on an instance of a bean with interceptors, the container intercepts the method invocation and invokes interceptors of the callback or business method.

The container identifies the first interceptor in the interceptor stack for the method. If no such interceptor exists, the container starts processing the decorator stack, as defined in Section A.5.8, “Decorator invocation”. Otherwise, the container builds an instance of `javax.interceptor.InvocationContext` and calls the appropriate interceptor method of the interceptor.

When any interceptor is invoked by the container, it may in turn call `InvocationContext.proceed()`. The container then identifies the first interceptor in the interceptor stack for the method such that the interceptor has not previously been invoked during this business method or lifecycle callback invocation. If no such interceptor exists, the container starts pro-

cessing the decorator stack. Otherwise, the container calls the appropriate interceptor method.

Eventually, by recursion, the interceptor stack is exhausted of uninvoked interceptors.

A.4. 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 {

    public Object load(Object id);
    public Object getId();

    public void save();
    public void delete();

    public Class getDataType();

}
```

The `DataAccessAuthorizationDecorator` class defines the authorization checks:

```
@Decorator
public abstract class DataAccessAuthorizationDecorator
    implements DataAccess {

    @Decorates DataAccess delegate;

    @LoggedIn User user;

    public void save() {
        authorize("save");
        delegate.save();
    }

    public void delete() {
        authorize("delete");
        delegate.delete();
    }

    private void authorize(String action) {
        try {
            Object id = delegate.getId();
            Class type = delegate.getDataType();
            if ( user.hasPermission(action, type, id) )
            {
                System.out.println("Authorized for " + action);
            }
            else {
                System.out.println("Not authorized for " + action);
                throw new NotAuthorizedException(action);
            }
        }
        catch (NotAuthenticatedException nae) {
            System.out.println("Not authenticated");
            throw nae;
        }
    }

}
```

The `@Decorator` annotation identifies the `DataAccessAuthorizationDecorator` class as a decorator. The `@Decorates` annotation identifies the *delegate attribute*, 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.

A.5. Decorators

A *decorator* implements one or more bean types and intercepts business method invocations for methods defined by the implemented bean types. These bean types are called *decorated types*.

A decorator is a managed bean. The set of decorated types of a decorator includes all interfaces implemented directly or indirectly by the bean class, 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.

Alternative definition: the set of decorated types includes all interfaces implemented directly and indirectly by both the decorator bean class and the declared type of the delegate attribute.

A decorator with scope `@Dependent` must be serializable. If a decorator has scope `@Dependent` and is not serializable, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

A.5.1. Declaring a decorator

A decorator is declared by annotating the bean class with the `@Decorator` stereotype.

```
@Decorator
class TimestampLogger implements Logger { ... }
```

A.5.2. Decorator delegate attributes

All decorators have a *delegate attribute*.

A delegate attribute is a non-static, non-final field of a decorator bean class.

The delegate attribute may be declared using the `@Decorates` annotation or `<Decorates>` element:

```
@Decorator
class TimestampLogger implements Logger {
    @Decorates Logger logger;
    ...
}
```

In this case, the decorator is bound to any bean that has the type of the delegate attribute as a bean type.

The declared type of the delegate attribute must be a Java interface type. If the declared type of a delegate attribute is not a Java interface type, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

The delegate may optionally declare one or more bindings:

```
@Decorator
class TimestampLogger implements Logger {
    @Decorates @Debug Logger logger;
    ...
}
```

In this case, the decorator is bound to any bean that has the type of the delegate attribute as a bean type, and declares all the bindings specified by the delegate attribute.

All delegate bindings must be explicitly declared. If no binding is explicitly declared by the delegate attribute, the set of bindings is empty.

A decorator must have exactly one delegate attribute. If a decorator has more than one delegate attribute, or does not have a delegate attribute, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a decorator applies to a managed bean, and the bean class is declared final, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

If a decorator applies to 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 definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

A.5.3. Decorated types of a decorator

A decorator is not required to implement all of the bean types of its delegate attribute. If a decorator does not implement a bean type of the delegate attribute, that API will not be intercepted by the decorator.

A decorator may be an abstract Java class, and is not required to implement all methods of its bean types. If a decorator does not implement a method of one of its bean types, that method will not be intercepted by the decorator.

The declared type of the decorator delegate attribute must implement or extend all of the decorated types of the decorator. If a decorator delegate attribute does not implement or extend a decorated type of the decorator, the container automatically detects the problem and treats it as a definition error, as defined in Section 10.3, “Problems detected automatically by the container”.

A.5.4. Decorator enablement and ordering

By default, decorators are not enabled. A decorator must be explicitly enabled by listing its bean class under the `<Decorators>` element in `beans.xml`.

```
<Decorators>
    <myfwk:TimestampLogger/>
    <myfwk:IdentityLogger/>
</Decorators>
```

The order of the decorator declarations determines the decorator ordering. Decorators which occur earlier in the list are called first.

If a class listed under the `<Decorators>` element is not the bean class of at least one decorator, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

If the bean class of a decorator with a disabled deployment type is listed under the `<Decorators>` element, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

If the `<Decorators>` element is specified in more than one `beans.xml` document, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 10.3, “Problems detected automatically by the container”.

Decorators are called after interceptors.

Would it be better to unify interceptors and decorators into a single stack, so that they can be interleaved?

A.5.5. The `Decorator` object for a decorator

The `Bean` object for a decorator must implement the interface `Decorator`.

```
public interface Decorator extends Bean<Object> {

        public Type getDelegateType();

        public Set<Annotation> getDelegateBindings();

        public void setDelegate(Object instance, Object delegate);

}
```

A.5.6. Decorator resolution

The following method returns the ordered list of enabled decorators for a set of bean types and a set of bindings.

```
public interface BeanManager {
```

```
    List<Decorator> resolveDecorators(Set<Type> types, Annotation... bindings);

    ...
}
```

The first argument is the set of bean types of the decorated bean. The annotations are bindings declared by the decorated bean.

If two instances of the same binding type are passed to `resolveDecorators()`, a `DuplicateBindingTypeException` is thrown.

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

If the set of bean types is empty, an `IllegalArgumentException` is thrown.

The following algorithm must be used by the container when resolving decorators:

- First, the container identifies the set of *matching* enabled decorators where the declared type of the delegate attribute is one of the given bean types. For this purpose, primitive types are considered to be identical to their corresponding wrapper types in `java.lang`, array types are considered identical only if their element types are identical and parameterized types are considered identical only if both the type and all type parameters are identical.
- Next, the container considers the given bindings. If no bindings were passed to `resolveDecorators()`, the container assumes the binding `@Current`. The container narrows the set of matching decorators to just those where for each binding declared by the decorator delegate attribute, there is a given binding with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding` (see Section 5.1.3, “Binding annotations with members”).
- Next, the container orders the matching decorators according to the decorator ordering specified in Section A.5.4, “Decorator enablement and ordering” and returns the resulting list of decorators. If no matching decorators exist in the set, an empty list is returned..

A.5.7. Decorator stack creation

When a bean with decorators is created, the container must:

- Identify the decorators for the bean by calling `BeanManager.resolveDecorators()` passing the bean types and bindings of the bean.
- For each decorator, call `BeanManager.getReference()`, passing the `Decorator` object, to obtain an instance of the decorator.
- For each returned decorator instance, call `Decorator.setDelegate()` to inject an object that implements the declared type of the delegate attribute to the delegate attribute of the decorator instance.
- Build an ordered list of the decorator instances.

The resulting ordered list of decorator instances is called the *decorator stack*.

A.5.8. 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 the interceptor stack, as defined in Section A.3.12, “Interceptor invocation”, invokes decorators of the bean.

The container searches for the first decorator in the decorator stack for 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 attribute. The container intercepts the delegate invocation and searches for the first decorator in the decorator stack for the instance such that:

- the decorator implements the method that is being invoked upon the delegate, and
- the decorator has not previously been invoked during this business method invocation.

If no such decorator exists, the container invokes the business method of the intercepted instance. Otherwise, the container calls the method of the decorator.

Eventually, by recursion, the decorator stack is exhausted of uninvoked decorators.

Appendix B. Helper literals

The Java language does not currently support a literal syntax for parameterized types or for inline instantiation of annotation values. Therefore, this specification defines helper classes to simplify these tasks.

B.1. Generic type literals

The following helper class allows inline instantiation of an object that represents a parameterized type.

```
public abstract class TypeLiteral<T> {

   private Type actualType;

   protected TypeLiteral() {
      Class<?> typeLiteralSubclass = getTypeLiteralSubclass(this.getClass());
      if (typeLiteralSubclass == null) {
         throw new RuntimeException(getClass() + " is not a subclass of TypeLiteral");
      }
      actualType = getTypeParameter(typeLiteralSubclass);
      if (actualType == null) {
         throw new RuntimeException(getClass() + " is missing type parameter in TypeLiteral");
      }
   }

   public final Type getType() {
      return actualType;
   }

   @SuppressWarnings("unchecked")
   public final Class<T> getRawType() {
      Type type = getType();
      if (type instanceof Class) {
         return (Class<T>) type;
      }
      else if (type instanceof ParameterizedType) {
         return (Class<T>) ((ParameterizedType) type).getRawType();
      }
      else if (type instanceof GenericArrayType) {
         return (Class<T>) Object[].class;
      }
      else {
         throw new RuntimeException("Illegal type");
      }
   }

   private static Class<?> getTypeLiteralSubclass(Class<?> clazz) {
      Class<?> superclass = clazz.getSuperclass();
      if (superclass.equals(TypeLiteral.class)) {
         return clazz;
      }
      else if (superclass.equals(Object.class)) {
         return null;
      }
      else {
         return (getTypeLiteralSubclass(superclass));
      }
   }

   private static Type getTypeParameter(Class<?> superclass) {
      Type type = superclass.getGenericSuperclass();
      if (type instanceof ParameterizedType) {
         ParameterizedType parameterizedType = (ParameterizedType) type;
         if (parameterizedType.getActualTypeArguments().length == 1) {
            return parameterizedType.getActualTypeArguments()[0];
         }
      }
      return null;
   }

}
```

An object that represents any parameterized type may be obtained by subclassing `TypeLiteral`.

```
TypeLiteral type = new TypeLiteral<List<String>>() {};
```

This object may be passed to APIs that perform typesafe resolution.

B.2. Annotation instance literals

The following helper class allows inline instantiation of annotation type instances.

```
public abstract class AnnotationLiteral<T extends Annotation>
        implements Annotation {

   private Class<T> annotationType;
   private Method[] members;

   protected AnnotationLiteral() {
      Class<?> annotationLiteralSubclass = getAnnotationLiteralSubclass(this.getClass());
      if (annotationLiteralSubclass == null) {
         throw new RuntimeException(getClass() + "is not a subclass of AnnotationLiteral ");
      }
      annotationType = getTypeParameter(annotationLiteralSubclass);
      if (annotationType == null) {
         throw new RuntimeException(getClass() + " is missing type parameter in AnnotationLiteral");
      }

      this.members = annotationType.getDeclaredMethods();
   }

   private static Class<?> getAnnotationLiteralSubclass(Class<?> clazz) {
      Class<?> superclass = clazz.getSuperclass();
      if (superclass.equals(AnnotationLiteral.class)) {
         return clazz;
      }
      else if (superclass.equals(Object.class)) {
         return null;
      }
      else {
         return getAnnotationLiteralSubclass(superclass);
      }
   }

   @SuppressWarnings("unchecked")
   private static <T> Class<T> getTypeParameter(Class<?> annotationLiteralSuperclass) {
      Type type = annotationLiteralSuperclass.getGenericSuperclass();
      if (type instanceof ParameterizedType) {
         ParameterizedType parameterizedType = (ParameterizedType) type;
         if (parameterizedType.getActualTypeArguments().length == 1) {
            return (Class<T>) parameterizedType
                  .getActualTypeArguments()[0];
         }
      }
      return null;
   }

   public Class<? extends Annotation> annotationType() {
      return annotationType;
   }

   @Override
   public String toString()  {
     String string = "@" + annotationType().getName() + "(";
     for (int i = 0; i < members.length; i++)
     {
        string += members[i].getName() + "=";
        string += invoke(members[i], this);
        if (i < members.length - 1)
        {
           string += ",";
        }
     }
     return string + ")";
   }

   @Override
   public boolean equals(Object other) {
      if (other instanceof Annotation) {
         Annotation that = (Annotation) other;
         if (this.annotationType().equals(that.annotationType())) {
            for (Method member : members) {
               Object thisValue = invoke(member, this);
               Object thatValue = invoke(member, that);
               if (!thisValue.equals(thatValue)) {
                  return false;
               }
            }
            return true;
         }
      }
      return false;
```

```
   }

   @Override
   public int hashCode() {
      int hashCode = 0;
      for (Method member : members) {
         int memberNameHashCode = 127 * member.getName().hashCode();
         int memberValueHashCode = invoke(member, this).hashCode();
         hashCode += memberNameHashCode ^ memberValueHashCode;
      }
      return hashCode;
   }

   private static Object invoke(Method method, Object instance) {
      try {
         method.setAccessible(true);
         return method.invoke(instance);
      }
      catch (IllegalArgumentException e) {
         throw new ExecutionException("Error checking value of member method " +
            method.getName() + " on " + method.getDeclaringClass(), e);
      }
      catch (IllegalAccessException e) {
         throw new ExecutionException("Error checking value of member method " +
            method.getName() + " on " + method.getDeclaringClass(), e);
      }
      catch (InvocationTargetException e) {
         throw new ExecutionException("Error checking value of member method " +
            method.getName() + " on " + method.getDeclaringClass(), e);
      }
   }

}
```

An instance of an annotation type may be obtained by subclassing `AnnotationLiteral`.

```
public abstract class PayByBinding
    extends AnnotationLiteral<PayBy>
    implements PayBy {}
```

```
PayBy payby = new PayByBinding() { public value() { return CHEQUE; } };
```

Annotation values are often passed to APIs that perform typesafe resolution.

Appendix C. Packages

The annotations and interfaces defined by this specification are divided into several packages in the `javax` namespace.

C.1. javax.annotation

The following annotations are defined in the package `javax.annotation`:

- `@NonBinding`
- `@Named`
- `@Stereotype`

C.2. javax.interceptor

The following annotations are defined in the package `javax.interceptor`:

- `@Interceptor`
- `@InterceptorBindingType`

C.3. javax.decorator

The package `javax.decorator` contains annotations relating to decorators.

- `@Decorator`
- `@Decorates`

C.4. javax.context

The package `javax.context` contains annotations and interfaces relating to contexts.

- `@ScopeType`
- `@ApplicationScoped`
- `@RequestScoped`
- `@SessionScoped`
- `@ConversationScoped`
- `@Dependent`
- `Context`
- `Contextual`
- `Conversation`
- `ContextNotActiveException`

C.5. javax.inject

The package `javax.inject` contains annotations and interfaces relating to bindings, deployment types and injection.

- `@BindingType`
- `@DeploymentType`
- `@Produces`
- `@Disposes`
- `@Specializes`
- `@Realizes`
- `@Initializer`
- `@New`
- `@Any`
- `@Current`
- `@Production`
- `@Standard`

- `Instance`
- `TypeLiteral`
- `AnnotationLiteral`

- `InjectionException`
- `UnsatisfiedResolutionException`
- `AmbiguousResolutionException`
- `UnproxyableResolutionException`
- `DuplicateBindingTypeException`
- `CreationException`
- `IllegalProductException`

C.6. javax.inject.spi

The package `javax.inject.spi` contains the integration SPI.

- `BeforeBeanDiscovery`
- `AfterBeanDiscovery`
- `AfterDeploymentValidation`
- `BeanManager`
- `Bean`
- `Interceptor`
- `Decorator`

- `InjectionPoint`
- `InterceptionType`

C.7. javax.event

The package `javax.event` contains annotations and interfaces relating to events.

- `@Observes`
- `@IfExists`
- `@Asynchronously`
- `@AfterTransactionCompletion`
- `@AfterTransactionFailure`
- `@AfterTransactionSuccess`
- `@BeforeTransactionCompletion`
- `Event`
- `Observer`
- `ObserverException`