JSR-299: Contexts and Dependency Injection for the Java EE platform

---

JSR-299 Expert Group

Specification lead
Gavin King, Red Hat Middleware, LLC

Version
Expert group draft

Table of Contents

Evaluation license vii
1. Architecture 1
1.1. Contracts 1
1.2. Relationship to other specifications 2
1.2.1. Relationship to the Java EE platform specification 2
1.2.2. Relationship to EJB 2
1.2.3. Relationship to managed beans 2
1.2.4. Relationship to JSF 3
1.3. Introductory examples 3
1.3.1. JSF example 3
1.3.2. EJB example 5
1.3.3. Java EE component environment example 5
1.3.4. Event example 5
1.3.5. Interceptor example 6
1.3.6. Decorator example 7
2. Concepts 9
2.1. Functionality provided by the container to the bean 9
2.2. Bean types 10
2.2.1. Legal bean types 10
2.2.2. Typecasting between bean types 10
2.3. Bindings 11
2.3.1. Built-in binding types 11
2.3.2. Defining new binding types 12
2.3.3. Declaring the bindings of a bean 12
2.3.4. Specifying bindings of an injected field 13
2.3.5. Specifying bindings of a method or constructor parameter 13
2.4. Scopes 13
2.4.1. Built-in scope types 14
2.4.2. Defining new scope types 14
2.4.3. Declaring the bean scope 14
2.4.4. Default scope 15
2.5. Deployment types 15
2.5.1. Built-in deployment types 15
2.5.2. Defining new deployment types 16
2.5.3. Declaring the deployment type of a bean 16
2.5.4. Default deployment type 17
2.5.5. Enabled deployment types 17
2.5.6. Deployment type precedence 18
2.6. Bean EL names 18
2.6.1. Declaring the bean EL name 18
2.6.2. Default bean EL names 18
2.6.3. Beans with no EL name 18
2.7. Stereotypes 19
2.7.1. Defining new stereotypes 19
2.7.1.1. Declaring the default scope and deployment type for a stereotype 19
2.7.1.2. Specifying interceptor bindings for a stereotype 19
2.7.1.3. Specifying name defaulting for a stereotype 20
2.7.1.4. Stereotypes with additional stereotypes 20
2.7.2. Declaring the stereotypes for a bean 20
2.7.3. Built-in stereotypes 20
3. Programming model 22
3.1. Managed beans 22
3.1.1. Which Java classes are managed beans? 22
3.1.2. Bean types of a managed bean 22
3.1.3. Declaring a managed bean 23
3.1.4. Bean constructors 23
3.1.4.1. Declaring a bean constructor 23

3.1.4.2. Bean constructor parameters 24
3.1.5. Specializing a managed bean 24
3.1.6. Default name for a managed bean 24
3.2. Session beans 24
3.2.1. EJB remove methods of session beans 25
3.2.2. Bean types of a session bean 25
3.2.3. Declaring a session bean 25
3.2.4. Specializing a session bean 25
3.2.5. Default name for a session bean 26
3.3. Producer methods 26
3.3.1. Bean types of a producer method 26
3.3.2. Declaring a producer method 27
3.3.3. Producer method parameters 27
3.3.4. Specializing a producer method 27
3.3.5. Disposer methods 28
3.3.6. Disposed parameter of a disposer method 28
3.3.7. Declaring a disposer method 28
3.3.8. Disposer method parameters 29
3.3.9. Disposer method resolution 29
3.3.10. Default name for a producer method 29
3.4. Producer fields 29
3.4.1. Bean types of a producer field 30
3.4.2. Declaring a producer field 30
3.4.3. Default name for a producer field 30
3.5. Resources 30
3.5.1. Declaring a resource 31
3.5.2. Bean types of a resource 31
3.5.3. The JTA TransactionManager 31
3.6. Message destinations 32
3.6.1. Bean types of a message destination 32
3.6.2. Declaring a message destination 32
3.7. Injected fields 33
3.7.1. Declaring an injected field 33
3.8. Initializer methods 33
3.8.1. Declaring an initializer method 34
3.8.2. Initializer method parameters 34
3.9. The default binding at injection points 34
3.10. Beans with the @New binding 35
4. Inheritance and specialization 36
4.1. Inheritance of type-level metadata 36
4.2. Inheritance of member-level metadata 37
4.3. Specialization 37
4.3.1. Direct and indirect specialization 38
4.3.2. Most specialized enabled bean for a bean 39
4.3.3. Inconsistent specialization 39
5. Dependency injection, lookup and EL 40
5.1. Inter-module accessibility 40
5.2. Typesafe resolution 40
5.2.1. Unsatisfied and ambiguous dependencies 41
5.2.2. Legal injection point types 41
5.2.3. Assignability of raw and parameterized types 41
5.2.4. Primitive types and null values 42
5.2.5. Binding annotations with members 42
5.2.6. Multiple bindings 42
5.3. EL name resolution 43
5.3.1. Ambiguous EL names 43
5.4. Client proxies 43
5.4.1. Unproxyable bean types 44
5.4.2. Client proxy invocation 44
5.5. Dependency injection 44
5.5.1. Injectable references 45
5.5.2. Injected reference validity 45

5.5.3. Injection using the bean constructor 45
5.5.4. Injection of fields and initializer methods 45
5.5.5. Destruction of dependent objects 46
5.5.6. Invocation of producer or disposer methods 46
5.5.7. Access to producer field values 46
5.5.8. Invocation of observer methods 46
5.5.9. Injection point metadata 47
5.6. Programmatic lookup 47
5.6.1. The Instance interface 48
5.6.2. The built-in Instance 49
5.6.3. Using AnnotationLiteral 49
5.7. Integration with Unified EL 50
6. Scopes and contexts 51
6.1. The Contextual interface 51
6.1.1. Instance creation 51
6.1.2. Instance destruction 51
6.2. The Context interface 52
6.3. Normal scopes and pseudo-scopes 53
6.4. Dependent pseudo-scope 53
6.4.1. Dependent scope lifecycle 53
6.4.2. Dependent objects 54
6.4.3. Dependent object destruction 54
6.5. Contextual instances and contextual references 54
6.5.1. The active context object for a scope 55
6.5.2. Contextual instance of a bean 55
6.5.3. Contextual reference for a bean 55
6.5.4. Contextual reference validity 56
6.6. Passivation and passivating scopes 56
6.6.1. Passivation capable beans 56
6.6.2. Passivation capable dependencies 56
6.6.3. Passivating scopes 57
6.6.4. Validation of passivation capable beans and dependencies 57
6.7. Context management for built-in scopes 58
6.7.1. Request context lifecycle 58
6.7.2. Session context lifecycle 58
6.7.3. Application context lifecycle 58
6.7.4. Conversation context lifecycle 59
6.7.5. The Conversation interface 60
7. Lifecycle of contextual instances 61
7.1. Restriction upon bean instantiation 61
7.2. Container invocations and interception 62
7.3. Lifecycle of contextual instances 62
7.3.1. Lifecycle of managed beans 62
7.3.2. Lifecycle of stateful session beans 63
7.3.3. Lifecycle of stateless session and singleton beans 63
7.3.4. Lifecycle of producer methods 63
7.3.5. Lifecycle of producer fields 63
7.3.6. Lifecycle of resources 64
7.3.7. Lifecycle of message destinations 64
8. Decorators 65
8.1. Decorator beans 65
8.1.1. Declaring a decorator 65
8.1.2. Decorator delegate injection points 65
8.1.3. Decorated types of a decorator 66
8.2. Decorator enablement and ordering 66
8.3. Decorator resolution 66
8.4. Decorator stack creation 67
8.5. Decorator invocation 67
9. Interceptor bindings 68
9.1. Interceptor binding types 68
9.1.1. Interceptor binding types with additional interceptor bindings 68
9.1.2. Interceptor bindings for stereotypes 68

9.2. Declaring the interceptor bindings of an interceptor 68
9.3. Binding an interceptor to a bean 69
9.4. Interceptor enablement and ordering 69
9.5. Interceptor resolution 70
9.5.1. Interceptors with multiple bindings 70
9.5.2. Interceptor binding types with members 71
10. Events 72
10.1. Event types and binding types 72
10.2. The Observer interface 72
10.3. Observer resolution 72
10.3.1. Assignability of type variables, raw and parameterized types 73
10.3.2. Event binding types with members 73
10.3.3. Multiple event bindings 74
10.4. Firing events and registering observers 74
10.4.1. The Event interface 74
10.4.2. The built-in Event 75
10.5. Observer methods 75
10.5.1. Event parameter of an observer method 76
10.5.2. Declaring an observer method 76
10.5.3. Observer method parameters 76
10.5.4. Conditional observer methods 76
10.5.5. Transactional observer methods 77
10.5.6. Asynchronous observer methods 77
10.6. Observer notification 77
10.6.1. Observer method notification 77
10.6.2. Observer method invocation context 78
10.7. JMS event mappings 78
11. Portable extensions 80
11.1. The Bean interface 80
11.1.1. The Decorator interface 80
11.1.2. The Interceptor interface 81
11.2. The BeanManager object 81
11.2.1. Obtaining a contextual reference for a bean 81
11.2.2. Obtaining an injectable reference 81
11.2.3. Obtaining a Bean by type 82
11.2.4. Obtaining a Bean by name 82
11.2.5. Obtaining the most specialized bean 82
11.2.6. Obtaining a passivation capable bean by identifier 82
11.2.7. Resolving an ambiguous dependency 82
11.2.8. Registering a Bean 83
11.2.9. Registering an Observer 83
11.2.10. Firing an event 83
11.2.11. Observer resolution 84
11.2.12. Decorator resolution 84
11.2.13. Interceptor resolution 84
11.2.14. Validating a dependency 84
11.2.15. Determining if an annotation is a binding type, scope type, stereotype or interceptor binding type 85
11.2.16. Discovering the enabled deployment types 85
11.2.17. Registering a Context 85
11.2.18. Obtaining the active Context for a scope 85
11.2.19. Obtaining the ELResolver 85
11.3. Alternative metadata sources 85
11.4. Helper objects for Bean implementations 86
11.4.1. InjectionTarget, Producer and Listener 87
11.4.2. ManagedBean and SessionBean 88
11.4.3. The ProducerBean interface 88
11.4.4. The ObserverMethod interface 89
11.5. Container lifecycle events 89
11.5.1. BeforeBeanDiscovery event 89
11.5.2. AfterBeanDiscovery event 90
11.5.3. AfterDeploymentValidation event 90

11.5.4. BeforeShutdown event 90
11.5.5. ProcessAnnotatedType event 91
11.5.6. ProcessInjectionTarget event 91
11.5.7. ProcessBean event 92
11.5.8. ProcessObserverMethod event 93
11.6. Activities 93
11.6.1. Current activity 94
12. Packaging and deployment 95
12.1. Bean deployment archives 95
12.2. Application initialization lifecycle 95
12.3. Bean discovery 95
12.4. Problems detected automatically by the container 96
A. Helper literals 98
A.1. Generic type literals 98
A.2. Annotation instance literals 99
B. Packages 101
B.1. javax.interceptor 101
B.2. javax.decorator 101
B.3. javax.enterprise.event 101
B.4. javax.enterprise.context 101
B.5. javax.enterprise.context.spi 101
B.6. javax.enterprise.inject 101
B.7. javax.enterprise.inject.stereotype 102
B.8. javax.enterprise.inject.deployment 102
B.9. javax.enterprise.inject.spi 102

Evaluation license

NOTICE

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

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

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;

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

(iii) includes the following notice:

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

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

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

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

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

TRADEMARKS

No right, para, or interest in or to any trademarks, service marks, or trade names of Red Hat Middleware LLC or Red Hat's licensors is granted hereunder. Java and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

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

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

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL RED HAT MIDDLEWARE LLC OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF RED HAT MIDDLEWARE LLC AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Red Hat Middleware LLC (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Red Hat Middleware LLC with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Red Hat Middleware LLC a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

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

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

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

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
- The ability to *decorate* injected components
- An *event notification* model
- A web *conversation* context in addition to the three standard web contexts defined by the Java Servlets specification
- An SPI allowing *portable extensions* to integrate cleanly with the Java EE environment

To take advantage of these facilities, the Java EE component developer provides additional component-level metadata 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. A portable extension may provide objects to be injected or obtain contextual instances using the dependency injection service. The framework may even raise and observe events using the event notification service.

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

1.1. Contracts

This specification defines the responsibilities of:

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

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

Chapter 2, *Concepts*, Chapter 3, *Programming model*, Chapter 4, *Inheritance and specialization*, Chapter 9, *Interceptor bindings*, Section 8.1, “Decorator beans” and Section 10.5, “Observer methods” define the programming model for Java EE components that take advantage of the services defined by this specification, the responsibilities of the component developer, and the annotations used by the component developer to specify metadata.

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

directly with the container.

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

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

1.2. Relationship to other specifications

An application developer creates 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.

1.2.1. Relationship to the Java EE platform specification

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

The Java EE platform specification defines a facility for injecting *resources* that exist in the *Java EE component 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.

1.2.2. Relationship to EJB

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

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

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

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

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

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

1.2.3. Relationship to managed beans

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

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

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

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

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

1.2.4. Relationship to 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.

1.3. Introductory examples

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

1.3.1. JSF example

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

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

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

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

```
@Model
public class Credentials {

    private String username;
    private String password;

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

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

}
```

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

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

```
@SessionScoped @Model
public class Login {

    @Current Credentials credentials;
    @Users 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 defined in Section 2.4.1, “Built-in scope types” is a *scope type* that specifies the lifecycle of instances of `Login`.

The `@Current` annotation defined in Section 2.3.1, “Built-in binding types” is a *binding type*. Applied to a field, it causes the `Credentials` bean to be injected into any contextual instance of `Login` created by the container.

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

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

Applied to a field, it causes the JPA `EntityManager` to be injected by the container.

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

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

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

```
@Model
public class DocumentEditor {

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

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

}
```

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

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

1.3.2. EJB example

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

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

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

1.3.3. Java EE component environment example

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

```
public class Databases {

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

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

}
```

The JPA `@PersistenceContext` annotation identifies the JPA persistence unit.

1.3.4. Event example

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

```
@SessionScoped @Model
public class Login {

    @Current Credentials credentials;
    @Users EntityManager userDatabase;

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

    private User user;

    public void login() {

        List<User> results = ... ;

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

```
        }

    }

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

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

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

}
```

The method `fire()` of the built-in bean of type `Event` defined in Section 10.4.1, “The Event interface” allows the application to fire events. Events consist of an *event object*—in this case the `User`—and event bindings. Event bindings—such as `@LoggedIn` and `@LoggedOut`—allow observers to specify which events of a certain type they are interested in.

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

```
@SessionScoped
public class Permissions {

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

    @Users EntityManager userDatabase;

    void onLogin(@Observes @LoggedIn User user) {
        permissions = new HashSet( userDatabase.createQuery(
            "select p from Permission p where p.user.username=:username")
            .setParameter("username", user.getUsername())
            .getResultList() );
    }

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


}
```

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

The `@Observes` annotation defined in Section 10.5.2, “Declaring an observer method” identifies the method with the annotated parameter as an *observer method* that is called by the container whenever an event matching the type and bindings of the annotated parameter is fired.

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

```
@Inherited
@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.

1.3.6. 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;

    @Current Set<Permission> permissions;

    public void save() {
        authorize("save");
        delegate.save();
    }
```

```
    public void delete() {
        authorize("delete");
        delegate.delete();
    }

    private void authorize(String action) {
        Object id = delegate.getId();
        Class type = delegate.getDataType();
        if ( permissions.contains( new Permission(action, type, id) ) ) {
            System.out.println("Authorized for " + action);
        }
        else {
            System.out.println("Not authorized for " + action);
            throw new NotAuthorizedException(action);
        }
    }

}
```

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

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

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

Chapter 2. Concepts

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

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

A bean comprises the following attributes:

- A (nonempty) set of bean types
- A (nonempty) set of 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 explicitly annotating the bean class, or by allowing them to be defaulted by the container, as specified in Chapter 3, *Programming model*. In certain other cases—for example, Java EE component environment resources, defined in Section 3.5, “Resources”—the developer provides only the annotations and the bean implementation is provided by the container.

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

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

2.1. Functionality provided by the container to the bean

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

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

2.2. Bean types

A bean type defines a client-visible type of the bean. A bean may have multiple bean types. For example, the following bean has 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 Section 3.1.2, “Bean types of a managed bean”, Section 3.2.2, “Bean types of a session bean”, Section 3.3.1, “Bean types of a producer method”, Section 3.4.1, “Bean types of a producer field”, Section 3.5.2, “Bean types of a resource” and Section 3.6.1, “Bean types of a message destination”.

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

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

2.2.1. Legal bean types

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

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

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

Note that certain additional restrictions are specified in Section 5.4.1, “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 contextual reference to a bean to any bean type of the bean which is a Java interface. However, the client may not in general typecast its contextual reference to an arbitrary concrete bean type of the bean. For example, if our managed bean was injected to the following field:

```
@Current Business biz;
```

Then the following typecast is legal:

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

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

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

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

A *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 rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”.

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

2.3.1. Built-in binding types

The three standard binding types are defined in the package `javax.enterprise.inject`.

Every bean has the built-in binding `@Any`, even if it does not explicitly declare this binding, except for beans with the built-in binding `@New` defined in Section 3.10, “Beans with the @New binding”.

If a bean does not explicitly declare a binding, the bean has exactly one additional binding, of type `@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.enterprise.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 3.7, “Injected fields”) to determine the bean that is injected, according to the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”.

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

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

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

2.4. Scopes

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

- *singletons*, such as EJB singleton beans, whose state is shared between all clients,
- *stateless objects*, such as servlets and stateless session beans, which do not contain client-visible state, or
- objects that must be 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 five standard scope types defined by this specification, all defined in the package `javax.enterprise.context`.

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

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.enterprise.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 {}
```

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

2.4.3. Declaring the bean scope

The 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 12.4, “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 12.4, “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 to resolve ambiguous dependencies and EL names, as specified in Section 5.2.1, “Unsatisfied and ambiguous dependencies” and Section 5.3.1, “Ambiguous EL names”.

The set of deployment types is extensible.

2.5.1. Built-in deployment types

There are two standard deployment types defined by this specification, both in the package `javax.enterprise.inject.deployment`.

- The default deployment type for beans which do not explicitly declare a deployment type is `@Production`.
- All standard beans defined by this specification, and provided by the container, are defined using the `@Standard` de-

ployment type. For example, the `Conversation` object defined in Section 6.7.4, “Conversation context lifecycle” and the `BeanManager` object defined in Section 11.2, “The BeanManager object” have this deployment type.

No bean may be declared with the `@Standard` deployment type unless explicitly required by this specification.

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.enterprise.inject.deployment.DeploymentType` meta-annotation.

Applications and portable extensions 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 portable extension 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 12.4, “Problems detected automatically by the container”.

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 producer method or field does not declare any stereotype with a declared default deployment type, then the default deployment type is the deployment type of the bean that declares the producer method or field.
- If any other kind of 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 be instantiated at runtime.

The container inspects the deployment type of each bean that exists in a particular deployment (see Section 12.3, “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>
        <type>javax.enterprise.inject.deployment.Production</type>
        <type>org.mycompany.myfwk.DaoFramework</type>
        <type>org.mycompany.site.Australian</type>
        <type>org.mycompany.myfwk.Mock</type>
    </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 problem and treats it as a deployment problem, as defined in Section 12.4, “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 to resolve ambiguous EL names and ambiguous dependencies, as specified in Section 5.2.1, “Unsatisfied and ambiguous dependencies” and Section 5.3.1, “Ambiguous EL names”.

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

The following strings are valid EL names:

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

```
orderManager
```

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

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

Bean EL names allow the direct use of beans in JSP or JSF pages, as defined in Section 5.7, “Integration with Unified EL”. For example, a bean with the name `products` could be used like this:

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

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

2.6.1. Declaring the bean EL name

To specify the EL name of a bean, the `@javax.enterprise.inject.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 EL name is assumed.

2.6.2. Default bean EL names

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

- A bean class or producer method or field of a bean declares a `@Named` annotation and no EL 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 an EL 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 Section 3.1.6, “Default name for a managed bean”, Section 3.2.5, “Default name for a session bean”, Section 3.3.10, “Default name for a producer method” and Section 3.4.3, “Default name for a producer field”.

2.6.3. Beans with no EL name

If `@Named` is not specified, by neither the bean nor its stereotypes, a bean has no EL 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, 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 bean 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.enterprise.inject.stereotype.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 12.4, “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 12.4, “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 12.4, “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 9.1.2, “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 12.4, “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. 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. Built-in stereotypes

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

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

In addition, the special-purpose `@Interceptor` and `@Decorator` stereotypes are defined in Chapter 9, *Interceptor bindings*.

Chapter 3. Programming model

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

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

An application or portable extension may provide other kinds of beans by implementing the interface `Bean` defined in Section 11.1, “The Bean interface” and registering the implementation with the container, as defined in Section 11.2.8, “Registering a Bean”.

3.1. Managed beans

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

If the bean class of a managed bean is annotated with both the `@Interceptor` and `@Decorator` stereotypes, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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 12.4, “Problems detected automatically by the container”.

If the managed bean class is a parameterized type, it must have scope `@Dependent`. If a managed bean with a parameterized bean class declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.1.1. Which Java classes are managed beans?

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

- It is not a non-static inner class.
- It is a concrete class, or is annotated `@Decorator`.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in `ejb-jar.xml`.
- It 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.

3.1.2. 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 5.4.1, “Unproxyable bean

types".

3.1.3. Declaring a managed bean

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

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

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

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.

3.1.4. 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.

3.1.4.1. 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 12.4, “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 12.4, “Problems detected automatically by the container”.

3.1.4.2. Bean constructor parameters

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

3.1.5. 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 12.4, “Problems detected automatically by the container”.

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

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

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

3.1.6. 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`.

3.2. 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 12.4, “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 12.4, “Problems detected automatically by the container”.

If the session bean class is a parameterized type, it must have scope `@Dependent`. If a session bean with a parameterized bean class declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.2.1. EJB remove methods of session beans

If a session bean is a stateful session bean:

- If the scope is `@Dependent`, the application *may* call any EJB remove method of 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 7.3.2, “Lifecycle of stateful session beans”.

3.2.2. Bean types of a session bean

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

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

3.2.3. Declaring a session bean

A session bean does not require any special annotations. 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 { ... }
```

3.2.4. Specializing a session bean

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

If the bean class of X does not directly extend the bean class of another session bean, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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 { ... }
```

3.2.5. Default name for a session bean

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

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

3.3. Producer methods

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

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

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

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

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

If a producer method return type contains a wildcard type parameter the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If the producer method return type is a parameterized type with a type variable, it must have scope `@Dependent`. If a producer method with a parameterized return type with a type variable declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If a producer method return type is a type variable the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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.

3.3.1. Bean types of a producer method

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

- If the return type is an interface, the 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 5.4.1, “Unproxyable bean types”.

3.3.2. Declaring a producer method

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

```
public class Shop {

   @Produces PaymentProcessor getPaymentProcessor() { ... }

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

}
```

A producer method may also specify scope, 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 12.4, “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 session bean, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.3.3. 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.class) Order order, @Selected Product product)
   {
       order.setProduct(product);
       return order;
   }

}
```

3.3.4. 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 *directly specializes* Y, as defined in Section 4.3, “Specialization”.

If the method is static or does not directly override another producer method, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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;
   }

   ...

}
```

3.3.5. Disposer methods

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

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

A bean may declare multiple disposer methods.

3.3.6. Disposed parameter of a disposer method

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

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

3.3.7. Declaring a disposer method

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

Bindings may be declared by annotating the disposed parameter:

```
public class UserDatabaseEntityManager {

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

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

}
```

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 12.4, “Problems detected automatically by the container”.

If a disposer 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 12.4, “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 session bean, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.3.8. Disposer method parameters

In addition to the disposed parameter, a disposer 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) { ... }
```

3.3.9. Disposer method resolution

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

- the disposer method must be by the same bean class as the producer method, and
- the disposed parameter must resolve to the producer method, according to the rules of typesafe resolution defined in Section 5.2, “Typesafe resolution”.

If there are multiple disposer methods for a single producer method, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If the disposed parameter of a disposer method does not resolve to any producer method declared by the bean class, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.3.10. 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`:

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

This producer method is named `paymentProcessor`:

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

3.4. Producer fields

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

A producer field must be a 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 5.4, “Client proxies”.

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

If a producer field type contains a wildcard type parameter the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If the producer field type is a parameterized type with a type variable, it must have scope `@Dependent`. If a producer field with a parameterized type with a type variable declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If a producer field type is a type variable the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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.

3.4.1. Bean types of a producer field

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

- If the field type is an interface, the 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 5.4.1, “Unproxyable bean types”.

3.4.2. Declaring a producer field

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

```
public class Shop {

   @Produces PaymentProcessor paymentProcessor = ....;

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

}
```

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

```
public class Shop {

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

}
```

3.4.3. 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`:

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

3.5. 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;
```

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

A resource may not have an EL name.

3.5.1. 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.

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

```
@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.

If the producer field declaration specifies an EL name, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If the resource, persistence context, persistence unit, remote EJB or web service in the Java EE component environment is not of the same type as the producer field declaration, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.5.2. Bean types of a resource

The bean types of a resource are the declared type of the field declaration, together with `java.lang.Object`.

3.5.3. The JTA `TransactionManager`

The container must provide a built-in resource with bean type `javax.transaction.TransactionManager` and binding `@Current`, representing a reference to the JTA transaction manager.

3.6. Message destinations

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 *message destination* is a bean that represents a message destination reference 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 object. If the application calls the `close()` method of the injected object, 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 message destination must have scope `@Dependent` and may not have an EL name.

3.6.1. Bean types of a message destination

The bean types of a message destination depend upon whether it represents a queue or topic.

- If the message destination represents a queue, the bean types are `Queue`, `QueueConnection`, `QueueSession`, `QueueReceiver` and `QueueSender`.
- If the message destination represents a topic, the bean types are `Topic`, `TopicConnection`, `TopicSession`, `TopicSubscriber` and `TopicPublisher`.

In addition, the supertypes `Session`, `MessageProducer`, `MessageConsumer`, `Connection` and `Destination` are bean types of any message destination.

3.6.2. Declaring a message destination

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

The `@Resource` annotation specifies the metadata needed to obtain the topic or queue from the component environment.

Each message destination declaration must also specify the JMS `ConnectionFactory` to be used, using the `@javax.inject.enterprise.ConnectionFactory` annotation.

```
class Jms {

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

```
}
```

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

Alternatively, the `@ConnectionFactory` may be specified at the class level:

```
@ConnectionFactory(@Resource(name="java:global/env/jms/ConnectionFactory"))
class Jms {

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

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

}
```

A field level `@ConnectionFactory` annotation overrides any `@ConnectionFactory` specified at the class level.

If no `@ConnectionFactory` is specified at either the class or field level, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If the producer field declaration specifies an EL name or any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If the message destination in the Java EE component environment is not of the same type as the producer field declaration, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

3.7. 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 5.5, “Dependency injection”, injected fields are initialized by the container before initializer methods are called, and before the `@PostConstruct` callback occurs.

If a field is a producer field, it is not an injected field.

3.7.1. 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;

}
```

3.8. 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 5.5, “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.

3.8.1. Declaring an initializer method

An initializer method may be declared by annotating the method `@javax.enterprise.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 12.4, “Problems detected automatically by the container”.

3.8.2. Initializer method parameters

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

3.9. 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();
   }

}
```

3.10. Beans with the `@New` binding

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

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

However, this second bean:

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

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

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

This bean is available even if the bean class declares a disabled deployment type.

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

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

Chapter 4. Inheritance and specialization

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

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

The two cases are quite dissimilar.

By default, Java implementation reuse is assumed. In this case, the producer, disposer 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, disposer 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, injected fields, initializer methods and lifecycle callback methods of the first bean are inherited by the second bean.

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

4.1. Inheritance of type-level metadata

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

- If X is annotated with a 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 defined by the Java Language Specification.)

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

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

- 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 what is defined in the Java Language Specification.)

Scope types and deployment types explicitly declared by X and inherited by Y from X take precedence over default scopes and default deployment types of stereotypes declared or inherited by Y.

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

- scope types should be declared `@Inherited`,
- 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.

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

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

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

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

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

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

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

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

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

- If X declares a non-static producer or disposer method `x()` then Y does not inherit this method unless Y is explicitly declared to specialize X.

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

- If X declares a non-static producer field `x` then Y does not inherit this field unless Y is explicitly declared to specialize X.

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

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

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

```
public class DaoClient<T> {

    @Current Dao<T> dao;
    ...

}
```

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

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

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>
        <type>javax.enterprise.inject.deployment.Production</type>
        <type>org.mycompany.myfwk.Mock</type>
    </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 or observer 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.

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 or observer method, the method will be called upon an instance of the first bean.

4.3.1. Direct and indirect specialization

The annotation `@javax.enterprise.inject.deployment.Specializes` is used to indicate that one bean *directly specializes* another bean, as defined in Section 3.1.5, “Specializing a managed bean”, Section 3.2.4, “Specializing a session bean” and Section 3.3.4, “Specializing a producer method”.

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

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

Then X will inherit the bindings and name of Y:

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

If X declares a name explicitly, using `@Named`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

Furthermore, X must have all the bean types of Y. If X does not support some bean type of Y, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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`.

4.3.2. Most specialized enabled bean for a bean

If, in a particular deployment, an enabled bean X specializes a bean Y and is not itself specialized by any other enabled bean, we call it the *most specialized enabled bean* for Y in that deployment.

Non-static producer methods, producer fields, disposer methods and observer methods of a bean are invoked upon an instance of the most specialized enabled bean that specializes the bean, as defined by Section 5.5.6, “Invocation of producer or disposer methods”, Section 5.5.7, “Access to producer field values” and Section 5.5.8, “Invocation of observer methods”.

4.3.3. Inconsistent specialization

If, in a particular deployment, either

- some enabled bean X specializes another enabled bean Y and the deployment type of X does not have a higher precedence than the deployment type of 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 12.4, “Problems detected automatically by the container”.

Chapter 5. Dependency injection, lookup and EL

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

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

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

In general, a bean type or bean 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 name in an EL expression, 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 where at least one bean participating in every circular chain of dependencies has a normal scope, as defined in Section 6.3, “Normal scopes and pseudo-scopes”. The container is not required to support circular chains of dependencies where every bean participating in the chain has a pseudo-scope.

5.1. Inter-module accessibility

Beans packaged in a certain Java EE module or library are available for injection, lookup and EL resolution to classes and JSP/JSF pages packaged in some other Java EE module or library if and only if the first module or library is required to be *accessible* to the other module or library by the Java EE platform specification.

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

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

5.2. Typesafe resolution

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

When resolving beans that are eligible for injection to an injection point, the container identifies the set of *matching* enabled beans which satisfy the following conditions:

- The bean has a bean type that matches the required type. For this purpose, primitive types are considered to match their corresponding wrapper types in `java.lang` and array types are considered to match only if their element types are identical. Parameterized and raw types are considered to match if they are identical or if the bean type is *assignable* to the required type, as defined in Section 5.2.3, “Assignability of raw and parameterized types”.
- The bean has the required bindings. If no required bindings were explicitly specified, the container assumes the required 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 `@javax.enterprise.inject.NonBinding` (see Section 5.2.5, “Binding annotations with members”).
- The bean class is required to be accessible to the class that owns the injection point, according to the class loading requirements of the Java EE platform specification.

5.2.1. Unsatisfied and ambiguous dependencies

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

An *ambiguous dependency* exists at an injection point when there are multiple enabled accessible beans with the bean type and bindings declared by the injection point.

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

When an ambiguous dependency exists, the container attempts to resolve the ambiguity by examining the deployment types of the matching beans, as defined in Section 2.5.6, “Deployment type precedence”. If there is exactly one bean whose deployment type has a higher precedence than the deployment types of every other matching bean, the container will select this bean, and the ambiguous dependency is called *resolvable*.

The container must validate all injection points of all enabled beans and other Java EE component classes supporting injection when the application is initialized to ensure that there are no unsatisfied or unresolvable ambiguous dependencies. If an unsatisfied or unresolvable ambiguous dependency exists, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

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

5.2.2. Legal injection point types

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

If an injection point type is a type variable, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

5.2.3. Assignability of raw and parameterized types

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

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

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

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

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

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

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

5.2.4. 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 have null values, 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 12.4, “Problems detected automatically by the container”.

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

5.2.5. Binding annotations with members

According to the rules 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 12.4, “Problems detected automatically by the container”.

5.2.6. Multiple bindings

According to the rules 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.

5.3. EL name resolution

The process of matching a bean to a name used in EL is called *name resolution*. Since there is no typing information available in EL, the container may consider only the EL name.

When resolving a bean by name, the container identifies the set of *matching* enabled beans which satisfy the following conditions:

- The bean has the given EL name.
- The bean class is required to be accessible to classes in the same bean deployment archive as the JSP or JSF page containing the EL expression, according to the class loading requirements of the Java EE platform specification.

Name resolution usually occurs at runtime.

5.3.1. Ambiguous EL names

An *ambiguous EL name* exists in an EL expression when there are multiple enabled accessible beans which have the given EL name.

When an ambiguous EL name exists, the container attempts to resolve the ambiguity by examining the deployment types of the matching beans, as defined in Section 2.5.6, “Deployment type precedence”. If there is exactly one bean whose deployment type has a higher precedence than the deployment types of every other matching bean, the container will select this bean, and the ambiguous name is called *resolvable*.

All unresolvable ambiguous EL names are detected by the container when the application is initialized. If, in a particular deployment, two beans are both accessible to classes in a certain bean deployment archive, according to the class loading requirements of the Java EE platform specification, and either:

- the two beans have the same EL name and the same deployment type, or
- the EL name of one bean is of the form `x.y`, where `y` is a valid bean EL name, and `x` is the EL name of the other bean,

the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

5.4. Client proxies

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

A contextual reference to a bean with a normal scope, as defined in Section 6.3, “Normal scopes and pseudo-scopes”, is not a direct reference to a contextual instance of the bean (the object returned by `Contextual.create()`). Instead, the contextual reference is a *client proxy* object. A client proxy implements/extends some or all of the 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 may be passivated, even when the bean itself may not be. Therefore the container must use a client proxy whenever a bean with normal scope is injected into a bean with a passivating scope, as defined in Section 6.6, “Passivation and passivating scopes”. (On the other hand, beans with scope `@Dependent` must be serialized along with their client.)

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

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

5.4.1. 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 12.4, “Problems detected automatically by the container”.

5.4.2. 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.

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

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

5.5. Dependency injection

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

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

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

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

- non-contextual instances of session beans (for example, session beans obtained by the application from JNDI or injected using `@EJB`),
- non-contextual instances of managed beans, and

- instances of any other Java EE component class supporting injection.

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.

5.5.1. Injectable references

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

- Identify a bean according to the rules defined in Section 5.2, “Typesafe resolution” and resolving ambiguities according to Section 5.2.1, “Unsatisfied and ambiguous dependencies”.
- Obtain a contextual reference for this bean and the type of the injection point according to Section 6.5.3, “Contextual reference for a bean”.

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

- The container is permitted to directly inject a contextual instance of the bean, as defined in Section 6.5.2, “Contextual instance of a bean”.
- If an incompletely initialized instance of the bean is registered with the current `CreationalContext`, as defined in Section 6.1, “The Contextual interface”, the container is permitted to directly inject this instance.

However, in performing these optimizations, the container must respect the rules of *injected reference validity*.

5.5.2. Injected reference validity

Injected references to a bean are *valid* for a certain period of time:

- A reference to a bean injected into a field, bean constructor or initializer method is valid until the object into which it was injected is destroyed.
- A reference to a bean injected into a producer method is valid until the producer method bean instance that is being produced is destroyed.
- A reference to a bean injected into a disposer method or observer method is valid until the invocation of the method completes.

The application should not invoke a method of an invalid reference. If the application invokes a method of an invalid injected reference, the behavior is undefined.

5.5.3. 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.

5.5.4. 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.

5.5.5. 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.

5.5.6. Invocation of producer or disposer methods

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

- If the method is static, the container must invoke the method.
- Otherwise, if the method is non-static, the container must:
    - Determine the most specialized enabled bean that specializes the bean which declares the method, as defined in Section 4.3.2, “Most specialized enabled bean for a bean”.
    - 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, as a business method invocation, as defined in Section 7.2, “Container invocations and interception”.

The container passes an injectable reference to each injected method parameter. The container is also responsible for destroying dependent objects created during this invocation, as defined in Section 6.4.3, “Dependent object destruction”.

5.5.7. 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 enabled bean that specializes the bean which declares the producer field, as defined in Section 4.3.2, “Most specialized enabled bean for a bean”.
    - 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.

5.5.8. Invocation of observer methods

When the container calls an observer method (defined in Section 10.5, “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 enabled bean that specializes the bean which declares the observer method, as defined in Section 4.3.2, “Most specialized enabled bean for a bean”.
    - 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, as a business method invocation, as defined in Sec-

tion 7.2, “Container invocations and interception”.

The container must pass the event object to the event parameter and an injectable instance to each injected method parameter. The container is also responsible for destroying dependent objects created during this invocation, as defined in Section 6.4.3, “Dependent object destruction”.

5.5.9. Injection point metadata

The interface `javax.enterprise.inject.spi.InjectionPoint` provides access to metadata about an injection point. An instance of `InjectionPoint` may represent an injected field or a parameter of a bean constructor, initializer method, producer method, disposer method or observer method.

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

- The `getBean()` method returns the `Bean` object representing the bean that defines the injection point. If the injection point does not belong to a bean, `getBean()` returns a null value.
- The `getType()` and `getBindings()` methods return the required type and required 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.enterprise.inject.spi.AnnotatedField` or `javax.enterprise.inject.spi.AnnotatedParameter`, depending upon whether the injection point is an injected field or a constructor/method parameter.
- The `isDelegate()` method returns `true` if the injection point is a decorator delegate injection point, and `false` otherwise.
- The `isTransient()` method returns `true` if the injection point is a transient field, and `false` otherwise.

Occasionally, a component with scope `@Dependent` needs to access metadata relating to the object into which it is injected. For example, the following producer method creates injectable `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. The built-in implementation must be a passivation capable dependency, as defined in Section 6.6.2, “Passivation capable dependencies”.

If a bean that declares any scope other than `@Dependent` has an injection point of type `InjectionPoint` and binding `@Current`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “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 12.4, “Problems detected automatically by the container”.

5.6. Programmatic lookup

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

used when:

- the bean type or 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.enterprise.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();
```

5.6.1. The `Instance` interface

The `Instance` interface provides a method for obtaining instances of beans with a specified combination of required type and bindings, and inherits the ability to iterate beans with that combination of required type and bindings 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);

}
```

For an injected `Instance`:

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

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

```
@Any Instance<PaymentProcessor> any;
```

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

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

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

If two instances of the same binding type are passed to `select()`, an `IllegalArgumentException` 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 must:

- Identify a bean that matches the required type and required bindings and is accessible to the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Section 5.2, “Typesafe resolution” and resolving ambiguities according to Section 5.2.1, “Unsatisfied and ambiguous dependencies”.
- If typesafe resolution results in an unsatisfied dependency, throw an `UnsatisfiedResolutionException`. If typesafe resolution results in an unresolvable ambiguous dependency, throw an `AmbiguousResolutionException`.
- Otherwise, obtain a contextual reference for the bean and the required type, as defined in Section 6.5.3, “Contextual reference for a bean”.

The `iterator()` method must:

- Identify the set of beans that match the required type and required bindings and are accessible to the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Section 5.2, “Typesafe resolution”.
- Return an `Iterator`, that iterates over the set of contextual references for the resulting beans and required type, as defined in Section 6.5.3, “Contextual reference for a bean”.

5.6.2. 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.

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

5.6.3. Using `AnnotationLiteral`

When the application calls `select()`, it may pass instances of binding annotation types. The helper class `javax.enterprise.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; } });
```

5.7. Integration with Unified EL

The container must provide a Unified EL `ELResolver` to the servlet engine and JSF implementation that resolves bean EL names using the rules of name resolution defined in Section 5.3, “EL name resolution” and resolving ambiguities according to Section 5.3.1, “Ambiguous EL names”.

- If a name used in an EL expression does not resolve to any bean, the `ELResolver` must return a null value.
- Otherwise, if a name used in an EL expression resolves to exactly one bean, the `ELResolver` must return a contextual instance of the bean, as defined in Section 6.5.2, “Contextual instance of a bean”.

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.

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.enterprise.context.spi.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, CreationalContext<T> creationalContext);

}
```

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

In particular, the `Bean` interface defined in Section 11.1, “The Bean interface” extends `Contextual`, so all beans are contextual types.

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

6.1.1. Instance creation

The method `Contextual.create()` 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 must be wrapped and rethrown as an (unchecked) `CreationException`.

The interface `javax.enterprise.context.spi.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);

    ...

}
```

The method `push()` registers the incompletely initialized instance.

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

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

6.1.2. Instance destruction

The method `Contextual.destroy()` is responsible for destroying instances of the type. In particular, it is responsible for

destroying all dependent objects of an instance.

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

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

The interface `javax.enterprise.context.spi.CreationalContext` provides an operation that allows the `destroy()` method to destroy dependent objects.

```
public interface CreationalContext<T> {

    ...

    public void release();

}
```

The method `release()` destroys all dependent objects of the instance which is being destroyed, as defined in Section 6.4.3, “Dependent object destruction”.

6.2. The `Context` interface

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

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

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

```
public interface Context {

    public Class<? extends Annotation> getScopeType();

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

    boolean isActive();

}
```

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

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

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

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

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

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

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

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

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

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 instances belonging to that context are destroyed by passing them to the `Contextual.destroy()` method.

Contexts with normal scopes must obey the following rule:

Suppose beans `A`, `B` and `Z` all have normal scopes. Suppose `A` has an injection point `x`, and `B` has an injection point `y`. Suppose further that both `x` and `y` resolve to bean `Z` according to the rules of typesafe resolution. If `a` is the current instance of `A`, and `b` is the current instance of `B`, then both `a.x` and `b.y` refer to the same instance of `Z`. This instance is the current instance of `Z`.

Any scope that is not a normal scope is called a *pseudo-scope*. The concept of a current instance is not well-defined in the case of a pseudo-scope.

All 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 `@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, disposer 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 the container is creating a contextual instance of a bean, injecting its dependencies or creating its decorators, or
- when the container is injecting dependencies of a Java EE component class supporting injection, or
- when a contextual instance of a bean with scope `@Dependent` is created by the container to receive a producer method, producer field, disposer method or observer method invocation, or
- while an observer or disposer method is invoked, or
- while a Unified EL expression is evaluated, or
- when `Instance.get()`, `Instance.iterator().next()` 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.

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 decorators with scope `@Dependent` are dependent objects of the bean instance they decorate.
- An instance of a bean with scope `@Dependent` injected into a field, bean constructor or initializer method is a dependent object of the bean or Java EE component class instance into which it was injected.
- An instance of a bean with scope `@Dependent` injected into a producer method is a dependent object of the producer method bean instance that is being produced.
- An instance of a bean with scope `@Dependent` obtained by direct invocation of an `Instance` or `BeanManager` is a dependent object of the instance of `Instance` or `BeanManager`.

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

Finally, the container is permitted to destroy any `@Dependent` scoped contextual instance at any time if the instance is no longer referenced by the application (excluding weak, soft and phantom references).

6.5. Contextual instances and contextual references

The `Context` object is the ultimate source of the contextual instances that underly contextual references.

6.5.1. The active context object for a scope

From time to time, the container must obtain an *active context object* for a certain scope type.

The container must search for an active instance of `Context` associated with the scope type.

- If no active context object exists for the scope type, the container throws a `ContextNotActiveException`.
- If more than one active context object exists for the given scope type, the container must throw an `IllegalStateException`.

If there is exactly one active instance of `Context` associated with the scope type, we say that the scope is *active*.

6.5.2. Contextual instance of a bean

From time to time, the container must obtain a *contextual instance* of a bean.

The container must:

- obtain the active context object for the bean scope, then
- obtain an instance of the bean by calling `Context.get()`, passing the `Bean` instance representing the bean and an instance of `CreationalContext`.

From time to time, the container attempts to obtain a *contextual instance of a bean that already exists*, without creating a new contextual instance.

The container must:

- 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`.

A contextual instance of any of the built-in kinds of bean defined in Chapter 3, *Programming model* is considered an internal container construct, and it is therefore not strictly required that a contextual instance of a built-in kind of bean directly implement the bean types of the bean. However, in this case, the container is required to transform its internal representation to an object that does implement the bean types expected by the application before injecting or returning a contextual instance to the application.

6.5.3. Contextual reference for a bean

From time to time, the container must obtain a *contextual reference* for a bean and a given bean type of the bean. A contextual reference implements the given bean type and all bean types of the bean which are Java interfaces. A contextual reference is not, in general, required to implement all concrete bean types of the bean.

- If the bean has a normal scope and the given bean type cannot be proxied by the container, as defined in Section 5.4.1, “Unproxyable bean types”, the container throws an `UnproxyableResolutionException`.
- If the bean has a normal scope, then the contextual reference for the bean is a client proxy, as defined in Section 5.4, “Client proxies”, created by the container, that implements the given bean type and all bean types of the bean which are Java interfaces.
- Otherwise, if the bean has a pseudo-scope, the container must obtain a contextual instance of the bean.

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.

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. Passivation and passivating scopes

The temporary transfer of the state of an idle object held in memory to some form of secondary storage is called *passivation*. The transfer of the passivated state back into memory is called *activation*.

6.6.1. Passivation capable beans

A bean is called *passivation capable* if the container is able to temporarily transfer the state of any idle instance to secondary storage.

- As defined by the EJB specification, all stateful session beans are passivation capable. Stateless and singleton session beans are not passivation capable.
- A managed bean is passivation capable if and only if the bean class is serializable.
- A producer method is passivation capable if and only if it never returns a value which is not passivation capable at runtime. A producer method with a return type that implements or extends serializable is passivation capable. A producer method with a return type that is declared final and does not implement serializable is not passivation capable.
- A producer field is passivation capable if and only if it never refers to a value which is not passivation capable at runtime. A producer field with a type that implements or extends serializable is passivation capable. A producer field with a type that is declared final and does not implement serializable is not passivation capable.

A custom implementation of `Bean` is passivation capable if it implements the interface `PassivationCapable`. An implementation of `Contextual` that is not a bean is passivation capable if it implements both `PassivationCapable` and `Serializable`.

```
public interface PassivationCapable {
    public String getId();
}
```

The `getId()` method must return a value that uniquely identifies the instance of `Bean` or `Contextual`. It is recommended that the string contain the package name of the class that implements `Bean` or `Contextual`.

6.6.2. Passivation capable dependencies

A bean is called a *passivation capable dependency* if any contextual reference for that bean is preserved when the object holding the reference is passivated and then activated.

The container must guarantee that:

- all session beans are passivation capable dependencies,
- all beans with normal scope are passivation capable dependencies,

- all passivation capable beans with scope `@Dependent` are passivation capable dependencies,
- all resources and message destinations are passivation capable dependencies, and
- the built-in beans of type `Instance`, `Event`, `InjectionPoint` and `BeanManager` are passivation capable dependencies.

A custom implementation of `Bean` is a passivation capable dependency if it implements `PassivationCapable` or if `getScopeType()` returns a normal scope type.

6.6.3. Passivating scopes

A *passivating scope* requires that:

- beans with the scope are passivation capable, and
- implementations of `Contextual` passed to any context object for the scope are passivation capable.

Passivating scopes must be explicitly declared `@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. No other built-in scopes are passivating scopes.

6.6.4. Validation of passivation capable beans and dependencies

For every bean which declares a passivating scope, and for every stateful session bean, the container must validate that the bean truly is passivation capable and that, in addition, its dependencies are passivation capable.

If a managed bean which declares a passivating scope:

- is not passivation capable, or
- has a non-transient injected field, bean constructor parameter or initializer method parameter that does not resolve to a passivation capable dependency,

then the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

If a stateful session bean:

- has a non-transient injected field, bean constructor parameter or initializer method parameter that does not resolve to a passivation capable dependency,

then the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

If a producer method declares a passivating scope and:

- the container is able to determine that it is not passivation capable by inspecting its return type, or
- has a parameter that does not resolve to a passivation capable dependency,

then the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

If a producer field declares a passivating scope and:

- the container is able to determine that it is not passivation capable by inspecting its type,

then the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

In some cases, the container is not able to determine whether a producer method or field is passivation capable. If a produ-

cer method or field which declares a passivating scope returns an unserializable object at runtime, the container must throw an `IllegalProductException`. If a producer method or field of scope `@Dependent` returns an unserializable object for injection into an injection point that requires a passivation capable dependency, the container must throw an `IllegalProductException`

For a custom implementation of `Bean`, the container calls `getInjectionPoints()` to determine the injection points, and `InjectionPoint.isTransient()` to determine whether the injected point is a transient field.

If a bean which declares a passivating scope type, or any stateful session bean, has a decorator which is not a passivation capable dependency, the container automatically detects the problem and treats it as a deployment problem, as defined in Section 12.4, “Problems detected automatically by the container”.

6.7. Context management for built-in scopes

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

The built-in context object is active during servlet, web service and EJB invocations, or in the case of the conversation context object, for JSF requests. For other kinds of invocations, a portable extension may define a custom context object for any or all of the built-in scopes. For example, a third-party web application framework might provide a conversation context object for the built-in conversation scope.

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 `@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 `@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, and
- when the disposer method or `@PreDestroy` callback of any bean with scope `@RequestScoped` or `@ConversationScoped` is called.

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 `@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,
- during any Java EE web service invocation,
- during any asynchronous observer method notification,
- during any remote method invocation of any EJB, during any asynchronous method invocation of any EJB, during any call to an EJB timeout method and during message delivery to any EJB message-driven bean, and
- when the disposer method or `@PreDestroy` callback of any bean with any normal scope other than `@ApplicationScoped` is called.

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 shut down.

6.7.4. Conversation context lifecycle

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

- during all standard lifecycle phases of any JSF faces or non-faces request, and
- when the disposer method or `@PreDestroy` callback of any bean with scope `@RequestScoped` is called.

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 beginning 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.

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 (a redirect resulting from a navigation rule or JSF `NavigationHandler`) is automatically propagated to the resulting non-faces request, and to any other subsequent request to the same URL. This is accomplished via use of a 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.

The *conversation timeout*, which may be specified by calling `Conversation.setTimeout()` is a hint to the container that a conversation should not be destroyed if it has been active within the last given interval in milliseconds.

If the propagated conversation cannot be restored, the container must associate the request with a new transient conversation and throw an exception of type `javax.context.NonexistentConversationException` from the restore view phase of the JSF lifecycle. The application may handle this exception using the JSF `ExceptionHandler`.

The container ensures that a long-running conversation may be associated with at most one request at a time, by blocking or rejecting concurrent requests. If the container rejects a request, it must associate the request with a new transient conversation and throw an exception of type `javax.context.BusyConversationException` from the restore view phase of the JSF lifecycle. The application may handle this exception using the JSF `ExceptionHandler`.

6.7.5. The `Conversation` interface

The container provides a built-in bean with bean type `Conversation`, scope `@RequestScoped`, deployment type `@Standard` and binding `@Current`, named `javax.enterprise.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);
   public boolean isTransient();
}
```

- `begin()` marks the current transient conversation long-running. A conversation identifier may, optionally, be specified. If no conversation identifier is specified, an identifier is generated by the container.
- `end()` marks the current long-running conversation transient.
- `getId()` returns the identifier of the current long-running conversation, or a null value if the current conversation is transient.
- `getTimeout()` returns the timeout, in milliseconds, of the current conversation.
- `setTimeout()` sets the timeout of the current conversation.
- `isTransient()` returns `true` if the conversation is marked transient, or `false` if it is marked long-running.

If `end()` is called, and the current conversation is marked transient, an `IllegalStateException` is thrown.

If `begin()` is called, and the current conversation is already marked long-running, an `IllegalStateException` is thrown.

If `begin()` is called with an explicit conversation identifier, and a long-running conversation with that identifier already exists, an `IllegalArgumentException` is thrown.

Chapter 7. Lifecycle of contextual instances

The lifecycle of a contextual instance of a bean is managed by the context object for the bean's scope, as defined in Chapter 6, *Scopes and contexts*.

Every bean in the system is represented by an instance of the `Bean` interface defined in Section 11.1, “The Bean interface”. This interface is a subtype of `Contextual`. To create and destroy contextual instances, the context object calls the `create()` and `destroy()` operations defined by the interface `Contextual`, as defined in Section 6.1, “The Contextual interface”.

7.1. Restriction upon bean instantiation

The managed bean and EJB specifications place very few programming restrictions upon the bean class of a bean. In particular, the class is a concrete class and is not required to implement any special interface or extend any special superclass. Therefore, bean classes are easy to instantiate and unit test.

However, if the application directly instantiates a bean class, instead of letting the container perform instantiation, the resulting instance is not managed by the container and is not a contextual instance as defined by Section 6.5.2, “Contextual instance of a bean”. Furthermore, the capabilities listed in Section 2.1, “Functionality provided by the container to the bean” will not be available to that particular instance. In a deployed application, it is the container that is responsible for instantiating beans and initializing their dependencies.

If the application requires more control over instantiation of a contextual instance, a producer method or field may be used. Any Java object may be returned by a producer method or field. It is not required that the returned object be a contextual reference for a bean. However, if the object is not a contextual reference for another bean, the object will be contextual instance of the producer method bean, and therefore available for injection into other objects and use in EL expressions, but the other capabilities listed in Section 2.1, “Functionality provided by the container to the bean” will not be available to the object.

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

```
@SessionScoped
public class PaymentStrategyProducer {

   private PaymentStrategyType paymentStrategyType;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      paymentStrategyType = type;
   }

   @Produces PaymentStrategy getPaymentStrategy(@CreditCard PaymentStrategy creditCard,
                                                @Cheque PaymentStrategy cheque,
                                                @Online PaymentStrategy online) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return creditCard;
         case CHEQUE: return cheque;
         case ONLINE: return online;
         default: throw new IllegalStateException();
      }
    }

}
```

In this case, any object returned by the producer method has already had its dependencies injected, receives lifecycle callbacks and event notifications and 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, any object returned by the producer method will not have any dependencies injected by the container, receives no lifecycle callbacks or event notifications and does not have interception enabled.

7.2. Container invocations and interception

When the container invokes a method of a bean, the invocation may or may not be treated as a *business method invocation*. If, and only if, the invocation is a business method invocation:

- it passes through the method interceptor and decorator stacks, and
- in the case of a session bean, it is subject to EJB services such as declarative transaction management, concurrency, security and asynchronicity, as defined by the EJB specification.

Otherwise, the invocation is treated as a normal Java method call and is not intercepted by the container.

- Invocations of initializer methods by the container are not business method invocations.
- Invocations of producer, disposer and observer methods by the container are business method invocations and are intercepted by method interceptors and decorators.
- Invocation of lifecycle callbacks by the container are not business method invocations, but are intercepted by interceptors for lifecycle callbacks.
- Invocations of interceptors and decorator methods during method or lifecycle callback interception are not business method invocations, and therefore no recursive interception occurs.

7.3. Lifecycle of contextual instances

The actual mechanics of bean creation and destruction varies according to what kind of bean is being created or destroyed.

7.3.1. Lifecycle of managed beans

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 5.5.3, “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 5.5.4, “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 5.5.5, “Destruction of dependent objects”.

7.3.2. Lifecycle of stateful session beans

When the `create()` method of a `Bean` object that represents a stateful session bean that is called, the container creates and returns a container-specific internal local reference to a new session bean instance. The reference must be passivation capable. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying stateful session bean instance. This object must be passivation capable.

When the `destroy()` method is called, and if the underlying EJB was not already removed by direct invocation of a remove method by the application, the container removes the stateful session bean. The `@PreDestroy` callback must be invoked by the container.

Note that the container performs additional work when the underlying EJB is created and removed, as defined in Section 5.5, “Dependency injection”

7.3.3. Lifecycle of stateless session and singleton beans

When the `create()` method of a `Bean` object that represents a stateless session or singleton session bean is called, the container creates and returns a container-specific internal local reference to the session bean. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying session bean. This object must be passivation capable.

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

Note that the container performs additional work when the underlying EJB is created and removed, as defined in Section 5.5, “Dependency injection”

7.3.4. Lifecycle of producer methods

When the `create()` method of a `Bean` object that represents a producer method is called, the container must invoke the producer method as defined by Section 5.5.6, “Invocation of producer or disposer methods”. The return value of the producer method, after method interception completes, is the new contextual instance to be returned by `Bean.create()`.

If the producer method returns a null value and the producer method bean has the scope `@Dependent`, the `create()` method returns a null value.

Otherwise, if the producer method returns a null value, and the scope of the producer method is not `@Dependent`, the `create()` method throws an `IllegalProductException`.

When the `destroy()` method is called, and if there is a disposer method for this producer method, the container must invoke the disposer method as defined by Section 5.5.6, “Invocation of producer or disposer methods”, passing the instance given to `destroy()` to the disposed parameter.

Finally, the container destroys dependent objects.

7.3.5. Lifecycle of producer fields

When the `create()` method of a `Bean` object that represents a producer field is called, the container must access the producer field as defined by Section 5.5.7, “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`.

7.3.6. Lifecycle of resources

When the `create()` method of a `Bean` object that represents a resource is called, the container creates and returns a container-specific internal reference to the Java EE component environment resource, entity manager, entity manager factory, remote EJB instance or web service reference. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying resource, entity manager, entity manager factory, remote EJB instance or web service reference. This object must be passivation capable.

When the `destroy()` method is called, the container discards this internal reference. For certain kinds of resource, for example persistence contexts, the container eventually performs additional cleanup to destroy state associated with the client or transaction, as required by the Java EE platform specification.

7.3.7. Lifecycle of message destinations

An instance of a message destination is a *proxy object*, provided by the container, that implements all the bean types specified in Section 3.6.1, “Bean types of a message destination”, delegating the actual implementation of these methods directly to the underlying JMS objects. This proxy object must be passivation capable.

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

When the `create()` method of a `Bean` object that represents a message destination is called, the container creates and returns a proxy object.

The methods of this proxy object delegate to JMS objects obtained as needed using the metadata provided by the message destination declaration.

- The `Destination` is obtained using the metadata specified by `@Resource`.
- The appropriate `ConnectionFactory` for the topic or queue is obtained using the metadata specified by `@ConnectionFactory`.
- 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()`.

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 message destination proxy object always throws an `UnsupportedOperationException`.

Chapter 8. Decorators

A *decorator* implements one or more bean types and intercepts business method invocations of beans which implement those bean types. These bean types are called *decorated types*.

Decorators are superficially similar to interceptors, but because they directly implement operations with business semantics, they are able to implement business logic and, conversely, unable to implement the orthogonal concerns for which interceptors are optimized.

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.

8.1. Decorator beans

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.

8.1.1. Declaring a decorator

A decorator is declared by annotating the bean class with the `@javax.decorator.Decorator` stereotype.

```
@Decorator
class TimestampLogger implements Logger { ... }
```

8.1.2. Decorator delegate injection points

All decorators have a *delegate injection point*.

A delegate injection point is an injection point of the bean class. The type and bindings of the injection point are called the *delegate type* and *delegate bindings*.

The delegate injection point must be be declared by annotating the field with the `@javax.decorator.Decorates` annotation:

```
@Decorator
class TimestampLogger implements Logger {
    @Decorates @Any Logger logger;
    ...
}
```

```
@Decorator
class TimestampLogger implements Logger {
    private Logger logger;

    public TimestampLogger(@Decorates @Debug Logger logger) {
        this.logger=logger;
    }
    ...
}
```

The decorator *applies* to any bean that is eligible for injection to the delegate injection point, according to the rules defined in Section 5.2, “Typesafe resolution”.

A decorator must have exactly one delegate injection point. If a decorator has more than one delegate injection point, or does not have a delegate injection point, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

The delegate injection point must be an injected field, initializer method parameter or bean constructor method parameter. If an injection point that is not an injected field, initializer method parameter or bean constructor method parameter is annotated `@Decorates`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

If a bean class that is not a decorator has an injection point annotated `@Decorates`, the container automatically detects the

problem and treats it as a definition error, as defined in Section 12.4, “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 12.4, “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 12.4, “Problems detected automatically by the container”.

The container must inject a *delegate* object to the delegate injection point. The delegate object implements the delegate type and delegates method invocations along the decorator stack.

8.1.3. Decorated types of a decorator

The delegate type of a decorator must implement or extend every decorated type. If the delegate type 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 12.4, “Problems detected automatically by the container”.

A decorator is not required to implement the delegate type.

A decorator may be an abstract Java class, and is not required to implement every method of every decorated type.

The decorator intercepts every method:

- declared by a decorated type of the decorator
- that is implemented by the bean class of the decorator.

8.2. 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`.

```
<beans>
    <decorators>
        <class>org.mycompany.myfwk.TimestampLogger</class>
        <class>org.mycompany.myfwk.IdentityLogger</class>
    </decorators>
</beans>
```

The order of the decorator declarations determines the decorator ordering. Decorators which occur earlier in the list are called first.

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 12.4, “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 12.4, “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 12.4, “Problems detected automatically by the container”.

Decorators are called after interceptors.

8.3. Decorator resolution

The following procedure must be used by the container when resolving decorators for a certain bean:

- First, the container identifies the set of *matching* enabled decorators such that the bean is eligible for injection to the delegate injection point according to the rules defined in Section 5.2, “Typesafe resolution”.

- Next, the container orders the matching decorators according to the decorator ordering specified in Section 8.2, “Decorator enablement and ordering”.

8.4. Decorator stack creation

When a bean with decorators is created, the container must:

- Identify the decorators for the bean according to Section 8.3, “Decorator resolution”.
- For each decorator, obtain a contextual reference, as defined in Section 6.5.3, “Contextual reference for a bean”.
- Build an ordered list of the decorator instances.

The resulting ordered list of decorator instances is called the *decorator stack*.

8.5. 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, 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. 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.

Chapter 9. Interceptor bindings

Managed beans and EJB session and message-driven beans support interception. The Java interceptors specification defines the basic programming model and semantics. This specification defines a typesafe mechanism for binding interceptors to beans using *interceptor binding types*.

9.1. Interceptor binding types

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 `@javax.interceptor.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.

9.1.1. 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 {}
```

Interceptor bindings are transitive—an interceptor binding declared by an interceptor binding type is inherited by all beans and other interceptor binding types that declare that interceptor binding type.

Interceptor binding types declared `@Target(TYPE)` may not be applied to interceptor binding types declared `@Target({TYPE, METHOD})`.

9.1.2. Interceptor bindings for stereotypes

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

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

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

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

9.2. Declaring the interceptor bindings of an interceptor

The interceptor bindings of an interceptor are specified by annotating the interceptor bean class with the binding types and the `@javax.interceptor.Interceptor` annotation.

```
@Transactional @Interceptor
public class TransactionInterceptor {

   @AroundInvoke
```

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

}
```

If an interceptor does not declare an `@Interceptor` annotation, it must be bound to beans using `@Interceptors` or `ejb-jar.xml`.

All interceptors declared using `@Interceptor` 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 12.4, “Problems detected automatically by the container”.

An interceptor for lifecycle callbacks may only declare interceptor binding types that are defined as `@Target(TYPE)`. If an interceptor for lifecycle callbacks declares an interceptor binding type that is defined `@Target({TYPE, METHOD})`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

9.3. Binding an interceptor to a bean

An interceptor bindings may be declared by annotating the bean class, or a method of the bean class, with the interceptor binding type.

In the following example, the `TransactionInterceptor` will be applied at the class level, and therefore applies to all business methods of the class:

```
@Transactional
public class ShoppingCart { ... }
```

In this example, the `TransactionInterceptor` will be applied at the method level:

```
public class ShoppingCart {

   @Transactional
   public void placeOrder() { ... }

}
```

9.4. Interceptor enablement and ordering

Interceptors may be enabled or disabled at deployment time. Disabled interceptors are never called at runtime.

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`.

```
<beans>
    <interceptors>
        <class>org.mycompany.myfwk.TransactionInterceptor</class>
        <class>org.mycompany.myfwk.LoggingInterceptor</class>
    </interceptors>
</beans>
```

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 12.4, “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 12.4, “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 12.4, “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.

9.5. Interceptor resolution

The following procedure must be used by the container when resolving interceptors for a certain bean and lifecycle callback or business method.

For a lifecycle callback, the *bean interceptor bindings* include the interceptor bindings declared by the bean at the class level, including interceptor bindings declared as meta-annotations of other interceptor bindings, recursively, and of stereotypes.

For a method, the bean interceptor bindings include the interceptor bindings declared by the bean at the class level, including interceptor bindings declared as meta-annotations of other interceptor bindings, recursively, and of stereotypes, together with all interceptor bindings declared at the method level, including interceptor bindings declared as meta-annotations of other interceptor bindings, recursively.

- First, the container identifies the set of *matching* enabled interceptors which satisfy the following conditions:
  - For each interceptor binding declared by the interceptor, there exists an interceptor binding in the set of bean interceptor bindings with (a) the same type and (b) the same annotation member value for each member which is not annotated `@javax.enterprise.inject.NonBinding` (see Section 9.5.2, “Interceptor binding types with members”).
  - 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 9.4, “Interceptor enablement and ordering”.

9.5.1. 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() { ... }

}
```

9.5.2. Interceptor binding types with members

According to the rules of interceptor resolution 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 12.4, “Problems detected automatically by the container”.

If the set of interceptor bindings of a bean or interceptor, including bindings inherited from stereotypes and other interceptor bindings, has two instances of a certain interceptor binding type and the instances have different values of some annotation member, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

Chapter 10. Events

Beans may produce and consume events. This facility allows beans to interact in a completely decoupled fashion, with no compile-time dependency between the two beans. Most importantly, it allows stateful beans in one architectural or physical tier of the application to synchronize their internal state with state changes that occur in a different tier.

An event comprises:

- A Java object—the *event object*
- A (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*.

10.1. Event types and binding types

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

An event type may not contain a type variable.

An event 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.enterprise.inject.BindingType` meta-annotation.

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

Any Java type may be an observed event type.

10.2. The `Observer` interface

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

Observers of events implement the interface `javax.enterprise.event.Observer` where `T` is the observed event type.

```
public interface Observer<T> {

    public boolean notify(T event);

}
```

If `notify()` returns `true`, the container may deregister the observer and stop sending event notifications to it. However, the container is permitted to call `notify()` even after a previous invocation of `notify()` has returned `false`.

The container is permitted to call `notify()` concurrently. Implementations of `Observer` are not permitted to perform their own concurrency management. It is therefore recommended that observers be immutable objects.

10.3. Observer resolution

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.

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 resolving observers of an event, the container identifies the set of observers which satisfy the following conditions:

- The event object must be assignable to the observed event type, taking type parameters into consideration.
- 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 `@javax.enterprise.inject.NonBinding`.

If the runtime type of the event object contains a type variable, the container must throw an `IllegalArgumentException`.

10.3.1. Assignability of type variables, raw and parameterized types

An event type is considered assignable to a type variable if the event type is assignable to the upper bound, if any.

A parameterized event type is considered assignable to a raw observed event type if the raw types are identical.

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

- the observed event type parameter is an actual type with identical raw type to the event type parameter, and, if the type is parameterized, the event type parameter is assignable to the observed event type parameter according to these rules, or
- the observed event type parameter is a wildcard and the event type parameter is assignable to the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the observed event type parameter is a type variable and the event type parameter is assignable to the upper bound, if any, of the type variable.

10.3.2. Event binding types with members

As usual, the binding type may have annotation members:

```
@BindingType
@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.

10.3.3. 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) { ... }
```

10.4. Firing events and registering observers

Beans fire events via an instance of the `javax.enterprise.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 `addObserver()` method registers an observer:

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

10.4.1. The `Event` interface

The `Event` interface provides a method for firing events with a specified combination of type and bindings, and a method for registering observers for events with that combination of type and bindings:

```
public interface Event<T> {

    public void fire(T event);
    public void addObserver(Observer<T> observer);
    public void removeObserver(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);
```

```
}
```

For an injected `Event`:

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

For example, this injected `Event` has specified type `LoggedInEvent` and specified binding `@Any`:

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

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

For example, this child `Event` has required type `AdminLoggedInEvent` and additional specified binding `@Admin`:

```
Event<AdminLoggedInEvent> admin = any.select(
            AdminLoggedInEvent.class,
            new AdminBinding() );
```

If the specified type contains a type variable, an `IllegalArgumentException` is thrown.

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

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

The method `fire()` fires an event with the specified bindings and notifies observers, as defined by Section 10.6, “Observer notification”.

If the runtime type of the event object contains a type variable, an `IllegalArgumentException` is thrown.

The method `addObserver()` registers the given observer for events of the specified type and bindings.

The method `removeObserver()` deregisters the given observer.

10.4.2. The built-in `Event`

The container must provide a built-in bean with:

- `Event<X>` in its set of bean types, for every Java type `X` that does not contain a type variable,
- every event 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.

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

10.5. 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 either a business method of the EJB or a static method of the bean class.

There may be arbitrarily many observer methods with the same event parameter type and bindings.

A bean may declare multiple observer methods.

10.5.1. 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.

The event parameter type may contain a type variable or wildcard.

10.5.2. Declaring an observer method

An observer method may be declared by annotating a parameter `@javax.enterprise.event.Observes`. That parameter is the event parameter. The declared type of the parameter is the observed event type.

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

If a method has more than one parameter annotated `@Observes`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

Observed event bindings may be declared by annotating the event parameter:

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

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 12.4, “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 12.4, “Problems detected automatically by the container”.

10.5.3. 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) { ... }
```

10.5.4. Conditional observer methods

A *conditional observer method* is an observer method which is notified of an event only if an instance of the bean that defines the observer method already exists in the current context.

A conditional observer method may be declared by specifying `notify=IF_EXISTS`.

```
public void refreshOnDocumentUpdate(@Observes(notify=IF_EXISTS) @Updated Document doc) { ... }
```

The enumeration `javax.enterprise.event.Notify` identifies the supported values of `notify`:

```
public enum Notify { IF_EXISTS, SYNCHRONOUSLY, ASYNCHRONOUSLY }
```

10.5.5. 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.

The enumeration `javax.enterprise.event.TransactionPhase` identifies the kind of transaction observer:

```
public enum TransactionPhase {
    IN_PROGRESS,
    BEFORE_COMPLETION,
    AFTER_COMPLETION,
    AFTER_FAILURE,
    AFTER_SUCCESS
}
```

A transactional observer method may be declared by specifying any value other than `IN_PROGRESS` for `during`:

```
void onDocumentUpdate(@Observes(during=AFTER_SUCCESS) @Updated Document doc) { ... }
```

10.5.6. Asynchronous observer methods

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

An asynchronous observer method may be declared by specifying `notify=ASYNCHRONOUSLY`.

```
void onDocumentUpdate(@Observes(notify=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 observer method is declared `notify=ASYNCHRONOUSLY` and `during=BEFORE_COMPLETION`, the container automatically detects the problem and treats it as a definition error, as defined in Section 12.4, “Problems detected automatically by the container”.

10.6. Observer notification

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

- determine the observers for that event according to the rules of observer resolution defined by Section 10.3, “Observer resolution”, 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.

10.6.1. Observer method notification

For every observer method of an enabled bean, the container is responsible for providing and registering an implementation of the `Observer` interface that delegates event notifications to the observer method, by calling the observer method as defined in Section 5.5.8, “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`.

10.6.2. Observer method invocation context

The transaction context, client security context and lifecycle contexts active when an observer method is invoked depend upon what kind of observer method it is.

- If the observer method is 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 before completion transactional observer method, it is called within the context of the transaction that is about to complete and with the same client security context and lifecycle contexts.
- Otherwise, if the observer method is any other kind of transactional observer method, it is called in an unspecified transaction context, but with the same client security context and lifecycle contexts as the transaction that just completed.
- Otherwise, the observer method is called in the same transaction context, client security context and lifecycle contexts as the invocation of `Event.fire()`.

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.

10.7. 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 message destination:

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

The event parameter specifies the *mapped event type and bindings*. Every message destination 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 12.4, “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 11. Portable extensions

A portable extension may integrate with the container by:

- Providing its own beans, interceptors and decorators to the container
- Injecting dependencies into its own objects using the dependency injection service
- Providing a context implementation for a custom scope
- Augmenting or overriding the annotation-based metadata with metadata from some other source

Bean definitions provided by a portable extension may be associated with a certain *activity*.

Integration with portable extensions is enabled via the important SPI interfaces `Bean` and `BeanManager`.

11.1. The `Bean` interface

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

```
public interface Bean<T>
        extends Contextual<T> {
    public Set<Type> getTypes();
    public Set<Annotation> getBindings();
    public Class<? extends Annotation> getScopeType();
    public Class<? extends Annotation> getDeploymentType();
    public String getName();
    public Class<?> getBeanClass();
    public boolean isNullable();
    public Set<InjectionPoint> getInjectionPoints();
}
```

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

- `getTypes()`, `getBindings()`, `getScopeType()`, `getDeploymentType()` and `getName()` must return the bean types, bindings, scope type, deployment type, and EL name of the bean, as defined in Chapter 2, *Concepts*.
- `getBeanClass()` returns the bean class of the managed bean or session bean or of the bean that declares the producer method or field.
- `getInjectionPoints()` returns a set of `InjectionPoint` objects, defined in Section 5.5.9, “Injection point metadata”, representing injection points of the bean, that will be validated by the container at initialization time.
- `isNullable()` must return `true` if the method `create()` sometimes returns a null value, and `false` otherwise, as defined in Section 5.2.4, “Primitive types and null values”.

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

An application or portable extension 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 message destinations) by implementing `Bean` and registering beans with the container, using the mechanism defined in Section 11.2.8, “Registering a Bean”.

11.1.1. The `Decorator` interface

The `Bean` object for a decorator must implement the interface `javax.enterprise.inject.spi.Decorator`.

```
public interface Decorator<T> extends Bean<T> {
    public Set<Type> getDecoratedTypes();
    public Type getDelegateType();
    public Set<Annotation> getDelegateBindings();
}
```

- `getDecoratedTypes()` returns the decorated types of the decorator.
- `getDelegateType()` and `getDelegateBindings()` return the delegate type and bindings of the decorator.

11.1.2. The `Interceptor` interface

The `Bean` object for an interceptor must implement `javax.enterprise.inject.spi.Interceptor`.

```
public interface Interceptor<T> extends Bean<T> {
    public Set<Annotation> getInterceptorBindings();
    public boolean intercepts(InterceptionType type);
    public Object intercept(InterceptionType type, T instance, InvocationContext ctx);
}
```

- `getInterceptorBindings()` returns the interceptor bindings of the interceptor.
- `intercepts()` returns `true` if the interceptor intercepts callbacks or business methods of the given type, and `false` otherwise.
- `intercept()` invokes the specified kind of lifecycle callback or business method upon the given instance.

An `InterceptionType` identifies the kind of lifecycle callback or business method.

```
public enum InterceptionType {
    AROUND_INVOKE, POST_CONSTRUCT, PRE_DESTROY, PRE_PASSIVATE, POST_ACTIVATE
}
```

11.2. The `BeanManager` object

Portable extensions sometimes interact directly with the container via programmatic API call. The interface `javax.enterprise.inject.spi.BeanManager` provides operations for obtaining contextual references for beans, along with many other operations of use to portable extensions.

The container provides a built-in bean with bean type `BeanManager`, scope `@Dependent`, deployment type `@Standard` and binding `@Current`. The built-in implementation must be a passivation capable dependency, as defined in Section 6.6.2, “Passivation capable dependencies”.

Thus, any bean may obtain an instance of `BeanManager` by injecting it:

```
@Current BeanManager manager;
```

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

11.2.1. Obtaining a contextual reference for a bean

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

```
public Object getReference(Bean<?> bean, Type beanType);
```

The first parameter is the `Bean` object representing the bean. The second parameter represents a bean type that must be implemented by any client proxy that is returned.

If the given type is not a bean type of the given bean, an `IllegalArgumentException` is thrown.

11.2.2. Obtaining an injectable reference

The method `BeanManager.getInjectableReference()` returns an injectable reference for a given injection point, as defined in Section 5.5.1, “Injectable references”.

If the `InjectionPoint` represents a decorator delegate injection point, `getInjectableReference()` returns a delegate, as defined in Section 8.1.2, “Decorator delegate injection points”.

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

If typesafe resolution results in an unsatisfied dependency, the container must throw an `UnsatisfiedResolutionException`. If typesafe resolution results in an unresolvable ambiguous dependency, the container must throw an `AmbiguousResolutionException`.

Implementations of `Bean` usually maintain a reference to an instance of `BeanManager`. When the `Bean` implementation performs dependency injection, it must obtain the contextual instances to inject by calling `BeanManager.getInjectableReference()`, passing an instance of `InjectionPoint` that represents the injection point and the instance of `CreationalContext` that was passed to `Bean.create()`.

11.2.3. Obtaining a `Bean` by type

The method `BeanManager.getBeans()` returns the set of beans which match the given required type and bindings and are accessible to the class into which the `BeanManager` was injected, according to the rules of typesafe resolution, as defined in Section 5.2, “Typesafe resolution”.

```
public Set<Bean<?>> getBeans(Type beanType, Annotation... bindings);
```

The first parameter is a required bean type. The remaining parameters are required bindings.

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

If the given type represents a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a binding type is given, an `IllegalArgumentException` is thrown.

11.2.4. Obtaining a `Bean` by name

The method `BeanManager.getBeans()` which accepts a string returns the set of beans which match the given EL name and are accessible to the class into which the `BeanManager` was injected, according to the rules of EL name resolution, as defined in Section 5.3, “EL name resolution”.

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

The parameter is an EL name.

11.2.5. 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, “Most specialized enabled bean for a bean”.

```
public <X> Bean<? extends X> getMostSpecializedBean(Bean<X> bean);
```

11.2.6. Obtaining a passivation capable bean by identifier

The method `BeanManager.getPassivationCapableBean()` returns the `PassivationCapableBean` with the given identifier.

```
public Bean<?> getPassivationCapableBean(String id);
```

11.2.7. Resolving an ambiguous dependency

The method `BeanManager.getHighestPrecedenceBean()` returns the `Bean` with the highest precedence deployment type in a set of beans.

```
public <X> Bean<? extends X> getHighestPrecedenceBean(Set<Bean<? extends X>> beans);
```

If there is no unique bean which has a higher precedence deployment type than all other beans in the set, the container must throw an `AmbiguousResolutionException`.

11.2.8. Registering a `Bean`

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

```
public void addBean(Bean<?> bean);
```

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

The container must fire a `ProcessBean` event containing the given `Bean` and then register the resulting `Bean` after all observers of this event have been notified. If the `Bean` object implements `ManagedBean`, the event should be of type `ProcessManagedBean`. If the `Bean` object implements `ProducerBean`, the event should be of type `ProcessProducerBean`.

11.2.9. Registering an `Observer`

The method `BeanManager.addObserver()` registers a new observer with the container. The container must begin sending event notifications to the registered observer.

```
public void addObserver(Observer<?> observer, Annotation... bindings);
```

The first parameter is the observer object. The observed event type is the actual type parameter of `Observer` declared by the class of the observer object. The remaining parameters are 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.

If the runtime type of the observer object contains a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a binding type is given, an `IllegalArgumentException` is thrown.

An alternative form fires a `ProcessObserverMethod` event and then registers an `ObserverMethod` with the container:

```
public void addObserver(ObserverMethod<?, ?> observerMethod);
```

The container must fire a `ProcessObserverMethod` event containing the `ObserverMethod` and then register the resulting `ObserverMethod` after all observers of this event have been notified.

The method `BeanManager.removeObserver()` deregisters an observer. The container must stop sending event notifications to the deregistered observer.

```
public interface BeanManager {

    public void removeObserver(Observer<?> observer);

    ...
}
```

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

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

11.2.10. Firing an event

The method `BeanManager.fireEvent()` fires an event and notifies observers, according to Section 10.6, “Observer notification”.

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

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

If the runtime type of the event object contains a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a binding type is given, an `IllegalArgumentException` is thrown.

11.2.11. Observer resolution

The method `BeanManager.resolveObservers()` resolves observers for an event according to the rules of observer resolution defined in Section 10.3, “Observer resolution”.

```
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 runtime type of the event object contains a type variable, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a binding type is given, an `IllegalArgumentException` is thrown.

11.2.12. Decorator resolution

The method `BeanManager.resolveDecorators()` returns the ordered list of enabled decorators for a set of bean types and a set of bindings, as defined in Section 8.3, “Decorator resolution”.

```
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 given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not a binding type is given, an `IllegalArgumentException` is thrown.

If the set of bean types is empty, an `IllegalArgumentException` is thrown.

11.2.13. Interceptor resolution

The method `BeanManager.resolveInterceptors()` returns the ordered list of enabled interceptors for a set of interceptor bindings, as defined in Section 9.5, “Interceptor resolution”.

```
List<Interceptor<?>> resolveInterceptors(InterceptionType type,
                                         Annotation... interceptorBindings);
```

If two instances of the same interceptor binding type are given, an `IllegalArgumentException` is thrown.

If no interceptor binding type instance is given, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not an interceptor binding type is given, an `IllegalArgumentException` is thrown.

11.2.14. Validating a dependency

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

```
public void validate(InjectionPoint injectionPoint);
```

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

11.2.15. Determining if an annotation is a binding type, scope type, stereotype or interceptor binding type

A portable extension may test an annotation to determine if it is a binding type, scope type, stereotype or interceptor binding type, obtain the set of meta-annotations declared by a stereotype or interceptor binding type, or obtain a `ScopeType` representing a scope type.

```
public boolean isScopeType(Class<? extends Annotation> annotationType);
public boolean isBindingType(Class<? extends Annotation> annotationType);
public boolean isInterceptorBindingType(Class<? extends Annotation> annotationType);
public boolean isStereotype(Class<? extends Annotation> annotationType);

public ScopeType getScopeDefinition(Class<? extends Annotation> scopeType)
public Set<Annotation> getInterceptorBindingTypeDefinition(Class<? extends Annotation> bindingType);
public Set<Annotation> getStereotypeDefinition(Class<? extends Annotation> stereotype);
```

11.2.16. Discovering the 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 List<Class<? extends Annotation>> getEnabledDeploymentTypes();
```

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

11.2.17. Registering a `Context`

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

```
public void addContext(Context context);
```

11.2.18. Obtaining the active `Context` for a scope

The method `BeanManager.getContext()` retrieves an active context object associated with the a given scope, as defined in Section 6.5.1, “The active context object for a scope”.

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

11.2.19. Obtaining the `ELResolver`

The method `BeanManager.getELResolver()` returns the `javax.el.ELResolver` specified in Section 5.7, “Integration with Unified EL”.

```
public ELResolver getELResolver();
```

11.3. Alternative metadata sources

A portable extension may provide an alternative metadata source, such as configuration by XML.

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

```
public interface AnnotatedType<X>
        extends Annotated {
    public Class<X> getJavaClass();
    public Set<AnnotatedConstructor<X>> getConstructors();
    public Set<AnnotatedMethod<? super X>> getMethods();
    public Set<AnnotatedField<? super X>> getFields();
}
```

```
public interface AnnotatedField<X>
        extends AnnotatedMember<X> {
    public Field getJavaMember();
}
```

```
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();
}
```

```
public interface AnnotatedMember<X>
        extends Annotated {
    public Member getJavaMember();
    public boolean isStatic();
    public AnnotatedType<X> getDeclaringType();
}
```

```
public interface AnnotatedCallable<X>
        extends AnnotatedMember<X> {
    public List<AnnotatedParameter<X>> getParameters();
}
```

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

```
public interface Annotated {
    public Type getType();
    public <T extends Annotation> T getAnnotation(Class<T> annotationType);
    public Set<Annotation> getAnnotations();
    public boolean isAnnotationPresent(Class<? extends Annotation> annotationType);
}
```

11.4. Helper objects for `Bean` implementations

Portable extensions sometimes provide custom implementations of `Bean` or inject dependencies directly into objects which are not contextual bean instances. To simplify the implementation of portable extensions, 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 5.5.3, “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);

    ...
}
```

- `createInjectionTarget()` returns a container provided instance of `InjectionTarget` for the given type or throws an `IllegalArgumentException` if there is a definition error associated with any injection point of the type.
- `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.

11.4.1. `InjectionTarget`, `Producer` and `Listener`

The interface `javax.enterprise.inject.spi.Producer` provides a generic operation for producing an instance of a type.

```
public interface Producer<T> {
    public T produce(CreationalContext<T> ctx);
    public Set<InjectionPoint> getInjectionPoints();
}
```

For any container provided implementation of `Producer` that represents a class:

- `produce()` calls the constructor annotated `@Initializer` if it exists, or the constructor with no parameters otherwise, as defined in Section 5.5.3, “Injection using the bean constructor”.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all injected fields, bean constructor parameters and initializer method parameters.

For any container provided implementation of `Producer` that represents a producer method or field:

- `produce()` calls the producer method on, or accesses the producer field of, a contextual instance of the most specialized bean that specializes the bean that declares the producer method, as defined in Section 5.5.6, “Invocation of producer or disposer methods”.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all parameters of the producer method.

The subinterface `javax.enterprise.inject.spi.InjectionTarget` provides operations for performing dependency injection and lifecycle callbacks on an instance of a type.

```
public interface InjectionTarget<T>
        extends Producer<T> {
    public void inject(T instance, CreationalContext<T> ctx);
    public void postConstruct(T instance);
    public void preDestroy(T instance);
}
```

For any container provided implementation of `InjectionTarget`:

- `inject()` performs dependency injection upon the given object. First, the container performs Java EE component environment injection according to the semantics required by the Java EE platform specification. Next, it sets the value all injected fields, and then calls all the initializer methods, as defined in Section 5.5.4, “Injection of fields and initializer methods”.
- `postConstruct()` calls the `@PostConstruct` callback, if it exists, according to the semantics required by the Java EE platform specification.
- `preDestroy()` calls the `@PreDestroy` callback, if it exists, according to the semantics required by the Java EE platform specification.

The interface `javax.enterprise.inject.spi.Listener` provides a generic operation for notifying an instance of a type.

```
public interface Listener<T> {
    public void notify(T instance);
    public Set<InjectionPoint> getInjectionPoints();
}
```

For any container provided implementation of `Listener` that represents a disposer method:

- `notify()` calls the disposer method on a contextual instance of the most specialized bean that specializes the bean that declares the disposer method, as defined in Section 5.5.6, “Invocation of producer or disposer methods”.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all parameters of the disposer method.

For any container provided implementation of `Listener` that represents an observer method:

- `notify()` calls the observer method on a contextual instance of the most specialized bean that specializes the bean that

declares the observer method, as defined in Section 5.5.8, “Invocation of observer methods”.

- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all parameters of the disposer method.

11.4.2. `ManagedBean` and `SessionBean`

The interface `javax.enterprise.inject.spi.BeanClass` defines common operations for managed and session beans.

```
public interface BeanClass<X> {
    public Set<ProducerBean<X, ?>> getProducerBeans();
    public Set<ObserverMethod<X, ?>> getObserverMethods();
    public InjectionTarget<X> getInjectionTarget();
    public void setInjectionTarget(InjectionTarget<X> injectionTarget);
    public Class<X> getBeanClass();
    public AnnotatedType<X> getAnnotatedType();
}
```

- `getProducerBeans()` returns a set of `ProducerBean` objects representing the producer methods and fields of the bean, including resources and message destinations.
- `getObserverMethods()` returns a set of `ObserverMethod` objects representing the observer methods of the bean.
- `getInjectionTarget()` initially returns a container provided implementation of `InjectionTarget` for the bean class.
- `setInjectionTarget()` allows a portable extension to replace the `InjectionTarget` used by this bean.
- `getBeanClass()` returns the bean class.
- `getAnnotatedType()` returns the `AnnotatedType` representing the bean class.

The interface `javax.enterprise.inject.spi.ManagedBean` represents a managed bean.

```
public interface ManagedBean<X>
        extends Bean<X>, BeanClass<X> {}
```

The interface `javax.enterprise.inject.spi.SessionBean` represents a session bean.

```
public interface SessionBean<X>
        extends Bean<Object>, BeanClass<X> {
    public String getEjbName();
    public SessionBeanType getSessionBeanType();
}
```

- `getEjbName()` returns the EJB name of the session bean.
- `getSessionBeanType()` returns a `javax.enterprise.inject.spi.SessionBeanType` representing the kind of session bean.

```
public enum SessionBeanType { STATELESS, STATEFUL, SINGLETON }
```

11.4.3. The `ProducerBean` interface

The `ProducerBean` interface represents a producer method or field.

```
public interface ProducerBean<X, T>
        extends Bean<T> {
    public Bean<X> getParentBean();
    public Producer<T> getProducer();
    public Listener<T> getDisposer();
    public void setProducer(Producer<T> producer);
    public void setDisposer(Listener<T> disposer);
    public Class<X> getBeanClass();
    public AnnotatedMember<? super X> getProducerMember();
    public AnnotatedParameter<? super X> getDisposedParameter();
}
```

- `getParentBean()` returns a container provided implementation of `Bean` for the bean class that has the producer method

or field.

- `getProducer()` initially returns a container-provided implementation of `Producer` representing the producer method or field.
- `getDisposer()` initially returns a container-provided implementation of `Listener` representing the disposer method, if any, or a null value.
- `setProducer()` replaces the `Producer` used by this bean.
- `setDisposer()` replaces the `Disposer` used by this bean.
- `getBeanClass()` returns the bean class of the bean that has the producer method or field.
- `getProducerMember()` returns the `AnnotatedMethod` or `AnnotatedField` representing the producer method or field.
- `getDisposedParameter()` returns the `AnnotatedParameter` representing the disposed parameter of the disposer method, if any, or a null value.

11.4.4. The `ObserverMethod` interface

The interface `javax.enterprise.inject.spi.ObserverMethod` represents an observer method.

```
public interface ObserverMethod<X, T>
        extends Observer<T> {
    public Bean<X> getParentBean();
    public Type getObservedEventType();
    public Set<Annotation> getObservedEventBindings();
    public Listener<T> getListener();
    public void setListener(Listener<T> listener);
    public AnnotatedParameter<? super X> getEventParameter();
}
```

- `notify()`, inherited from `Observer`, performs event notification, as defined in Section 10.6.1, “Observer method notification”.
- `getParentBean()` returns a container provided implementation of `Bean` for the bean that has the observer method.
- `getObservedEventType()` and `getObservedEventBindings()` return the observed event type and bindings of the observer method.
- `getListener()` initially returns a container-provided implementation of `Listener` representing the observer method.
- `setListener()` replaces the `Listener` used to invoke the observer method.

11.5. Container lifecycle events

During the application initialization process, the container fires a series of events, allowing portable extensions to integrate with the container initialization process defined in Section 12.2, “Application initialization lifecycle”.

Observers of these events must be service providers of the service `javax.enterprise.inject.spi.Extension` declared in `META-INF/services`.

```
public interface Extension {}
```

The container instantiates a single instance of each service provider at the beginning of the application initialization process and maintains a reference to it until the application shuts down.

For each service provider, the container must provide a bean of scope `@ApplicationScoped`, deployment type `@Standard`, binding `@Current`, supporting injection of a reference to the service provider instance. The bean types of this bean include the class of the service provider and all superclases and interfaces.

11.5.1. `BeforeBeanDiscovery` event

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

```
public interface BeforeBeanDiscovery {
    public void addBindingType(Class<? extends Annotation> bindingType);
    public void addScopeType(Class<? extends Annotation> scopeType, boolean normal, boolean passivating);
    public void addStereotype(Class<? extends Annotation> stereotype, Annotation... stereotypeDef);
    public void addInterceptorBindingType(Class<? extends Annotation> bindingType, Annotation... bindingTypeDef)
}
```

The operations of the `BeforeBeanDiscovery` instance allow a portable extension to declare that any annotation as a binding type, scope type, stereotype or interceptor binding type.

```
void beforeBeanDiscovery(@Observes BeforeBeanDiscovery event) { ... }
```

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

11.5.2. `AfterBeanDiscovery` event

The container must fire a second event when it has fully completed the bean discovery process, validated that there are no definition errors relating to the discovered beans, and registered `Bean` and `Observer` objects for the discovered beans, but before detecting deployment problems.

The event object must be of type `javax.enterprise.inject.spi.AfterBeanDiscovery`:

```
public interface AfterBeanDiscovery {
    public void addDefinitionError(Throwable t);
}
```

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

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

A portable extension 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.

11.5.3. `AfterDeploymentValidation` event

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

The event object must be of type `javax.enterprise.inject.spi.AfterDeploymentValidation`:

```
public interface AfterDeploymentValidation {
    public void addDeploymentProblem(Throwable t);
}
```

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

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

If any observer method of the `AfterDeploymentValidation` event throws an exception, the exception is treated as a deployment problem by the container.

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

11.5.4. `BeforeShutdown` event

The container must fire a final event after it has finished processing requests and destroyed all contexts.

The event object must be of type `javax.enterprise.inject.spi.BeforeShutdown`:

```
public interface BeforeShutdown {}
```

```
void beforeShutdown(@Observes BeforeShutdown event, BeanManager manager) { ... }
```

If any observer method of the `BeforeShutdown` event throws an exception, the exception is ignored by the container.

11.5.5. `ProcessAnnotatedType` event

The container must fire an event for each Java class it discovers, before it reads the declared annotations.

The event object must be of type `javax.enterprise.inject.spi.ProcessAnnotatedType<X>`, where `X` is the class.

```
public interface ProcessAnnotatedType<X> {
    public AnnotatedType<X> getAnnotatedType();
    public void setAnnotatedType(AnnotatedType<X> type);
}
```

- `getAnnotatedType()` returns the `AnnotatedType` object that will be used by the container to read the declared annotations.
- `setAnnotatedType()` replaces the `AnnotatedType`.

Any observer of this event is permitted to wrap and/or replace the `AnnotatedType`. The container must use the final value of this property, after all observers have been called, to read the annotations of this class.

For example, the following observer decorates the `AnnotatedType` for every class that is discovered by the container.

```
<T> void decorateAnnotatedType(@Observes ProcessAnnotatedType<T> pat) {
    pat.setAnnotatedType( decorate( pat.getAnnotatedType() ) );
}
```

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

11.5.6. `ProcessInjectionTarget` event

The container must fire an event for each managed bean, session bean, Java EE component class supporting injection.

The event object must be of type `javax.enterprise.inject.spi.ProcessInjectionTarget<X>`, where `X` is the managed bean class, session bean class or Java EE component class supporting injection.

```
public interface ProcessInjectionTarget<X> {
    public AnnotatedType<X> getAnnotatedType();
    public InjectionTarget<X> getInjectionTarget();
    public void setInjectionTarget(InjectionTarget<X> injectionTarget);
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotatedType()` returns the `AnnotatedType` representing the managed bean class, session bean class or other Java EE component class supporting injection.
- `getInjectionTarget()` returns the `InjectionTarget` object that will be used by the container to perform injection.
- `setInjectionTarget()`replaces the `InjectionTarget`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `InjectionTarget`. The container must use the final value of this property, after all observers have been called, whenever it performs injection upon the managed bean, session

bean or other Java EE component class supporting injection.

For example, this observer decorates the `InjectionTarget` for the all servlets.

```
<T extends Servlet> void decorateServlet(@Observes ProcessInjectionTarget<T> pit) {
    pit.setInjectionTarget( decorate( pit.getInjectionTarget() ) );
}
```

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

11.5.7. `ProcessBean` event

The container must fire an event for each bean it discovers, before registering the `Bean` object.

The event object must be of type `javax.enterprise.inject.spi.ProcessBean`.

- For a managed bean with bean class `X`, the container must raise an event of type `ProcessManagedBean<X>`.
- For a session bean with bean class `X`, the container must raise an event of type `ProcessSessionBean<X>`.
- For a producer method or field of a bean class `X` with method return type or field type `T`, the container must raise an event of type `ProcessProducerBean<X, T>`.

Resources and message destinations are considered to be producer fields.

```
public interface ProcessBean<X> {
    public Bean<X> getBean();
    public void setBean(Bean<X> bean);
    public void veto();
    public void addDefinitionError(Throwable t);
}
```

- `getBean()` returns the `Bean` object that is about to be registered.
- `setBean()` replaces the `Bean` object.
- `veto()` vetoes registration of the `Bean`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

```
public interface ProcessManagedBean<X>
        extends ProcessBean<X> {
    public ManagedBean<X> getManagedBean();
}
```

- `getManagedBean()` returns the `ManagedBean` representing the managed bean class.

```
public interface ProcessSessionBean<X>
        extends ProcessBean<X> {
    public SessionBean<X> getSessionBean();
}
```

- `getSessionBean()` returns the `SessionBean` representing the session bean class.

```
public interface ProcessProducerBean<X, T>
        extends ProcessBean<T> {
    public ProducerBean<X, T> getProducerBean();
}
```

- `getProducerBean()` returns the `ProducerBean` representing the producer method or field.

Any observer of this event is permitted to wrap and/or replace the `Bean`. When all observers have been called, the container registers the final value of this property, unless an observer called `veto()` or `addDefinitionError()`. The container must not register any `Bean` object if any observer called `veto()` or `addDefinitionError()`.

For example, the following observer decorates all session beans as they are discovered by the container.

```
<T> void decorateBean(@Observes ProcessSessionBean<T> pb) {
    pb.setBean( decorate( pb.getBean() ) );
}
```

This observer decorates the bean with bean class `PaymentProcessor`.

```
void decoratePaymentProcessor(@Observes ProcessBean<PaymentProcessor> pb) {
    pb.setBean( decorate( pb.getBean() ) );
}
```

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

11.5.8. `ProcessObserverMethod` event

The container must fire an event for each observer method it discovers, before registering the `Observer` object.

The event object must be of type `javax.enterprise.inject.spi.ProcessObserverMethod<X, T>` where `X` is the bean class of the bean that has the observer method and `T` is the observed event type.

```
public interface ProcessObserverMethod<X, T> {
    public ObserverMethod<X, T> getObserverMethod();
    public void setObserverMethod(ObserverMethod<X, T> observerMethod);
    public void veto();
    public void addDefinitionError(Throwable t);
}
```

- `getObserverMethod()`returns the `ObserverMethod` object that is about to be registered.
- `setObserverMethod()`replaces the `ObserverMethod`.
- `veto()` vetoes registration of the `Bean`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `ObserverMethod`. When all observers have been called, the container registers the final value of this property, unless an observer called `veto()` or `addDefinitionError()`. The container must not register any `ObserverMethod` object if any observer called `veto()` or `addDefinitionError()`.

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

11.6. 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.

The method `createActivity()` creates a new child activity of an activity and returns a `BeanManager` for the 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 non-contextual objects,
- 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 12.4, “Problems detected automatically by the container”.

The container is not required to support registration of an interceptor or decorator with a child activity. Portable extensions and applications should not depend upon the ability to register an interceptor or decorator with a child activity.

11.6.1. 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 5.7, “Integration with Unified EL”), all calls to any injected `BeanManager` object or `BeanManager` object obtained via JNDI lookup (as defined by Section 11.2, “The BeanManager object”), all calls to any injected `Event` object (as defined in Section 10.4.1, “The Event interface”) and all calls to any injected `Instance` object (as defined by Section 5.6.1, “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 extensions 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 12. Packaging and deployment

When an application is started, the container must perform *bean discovery*, detect definition errors and deployment problems and raise events that allow portable extensions to integrate with the deployment lifecycle.

Bean discovery is the process of determining:

- What beans, interceptors and decorators *exist* in the bean 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

Additional beans may be registered programmatically with the container by the application or a portable extension after the automatic bean discovery completes. Portable extensions may even integrate with the process of building the `Bean` object for a bean, to enhance the container's built-in functionality.

12.1. Bean deployment archives

Bean classes must be deployed in any 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). Any such archive or directory is called a *bean deployment archive*.

When searching for beans, the container considers:

- any Java class in any bean deployment archive, and
- any `ejb-jar.xml` file in any bean deployment archive.

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.

12.2. Application initialization lifecycle

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

- First, the container must search for service providers for the service `javax.enterprise.inject.spi.Extension` defined in Section 11.5, “Container lifecycle events”, instantiate a single instance of each service provider, and search the service provider class for observer methods of initialization events.
- Next, the container must fire an event of type `BeforeBeanDiscovery`, as defined in Section 11.5.1, “BeforeBeanDiscovery event”.
- Next, the container must perform bean discovery, and abort initialization of the application if any definition errors exist, as defined in Section 12.4, “Problems detected automatically by the container”.
- Next, the container must fire an event of type `AfterBeanDiscovery`, as defined in Section 11.5.2, “AfterBeanDiscovery event”, and abort initialization of the application if any observer registers a definition error.
- Next, the container must detect deployment problems by validating bean dependencies and specialization and abort initialization of the application if any deployment problems exist, as defined in Section 12.4, “Problems detected automatically by the container”.
- Next, the container must fire an event of type `AfterDeploymentValidation`, as defined in Section 11.5.3, “AfterDeploymentValidation event”, and abort initialization of the application if any observer registers a deployment problem.
- Finally, the container begins directing requests to the application.

12.3. Bean discovery

The container automatically discovers managed beans (according to the rules of Section 3.1.1, “Which Java classes are managed beans?”) and session beans in bean deployment archives and searches the bean classes for producer methods, producer fields, disposer methods and observer methods.

For each Java class in any bean deployment archive, the container must:

- create an `AnnotatedType` representing the class and fire an event of type `ProcessAnnotatedType`, as defined in Section 11.5.5, “ProcessAnnotatedType event”, and then
- inspect the class metadata to determine if it is a bean or other Java EE component class supporting injection, and then
- detect definition errors by validating the class and its metadata, and then
- if the class is a managed bean, session bean, or other Java EE component class supporting injection, create an `InjectionTarget` for the class, as defined in Section 11.4.1, “InjectionTarget, Producer and Listener”, and fire an event of type `ProcessInjectionTarget`, as defined in Section 11.5.6, “ProcessInjectionTarget event”, and then
- if the class is a bean, create a `Bean` object that implements the rules defined in Section 7.3.1, “Lifecycle of managed beans”, Section 7.3.2, “Lifecycle of stateful session beans” or Section 7.3.3, “Lifecycle of stateless session and singleton beans”, and fire an event of type `ProcessManagedBean` or `ProcessSessionBean`, as defined in Section 11.5.7, “ProcessBean event”.

For each session bean declared in `ejb-jar.xml` in any bean deployment archive the container must:

- create a `Bean` object that implements the rules defined in Section 7.3.2, “Lifecycle of stateful session beans” or Section 7.3.3, “Lifecycle of stateless session and singleton beans”, and fire an event of type `ProcessSessionBean`.

For each bean, the container must search the class for producer methods and fields, including resources and message destinations, and for each producer method or field:

- create a `Bean` object that implements the rules defined in Section 7.3.4, “Lifecycle of producer methods”, Section 7.3.5, “Lifecycle of producer fields”, Section 7.3.6, “Lifecycle of resources” or Section 7.3.7, “Lifecycle of message destinations”, and fire an event of type `ProcessProducerBean`, as defined in Section 11.5.7, “ProcessBean event”.

For each bean, the container must search the class for observer methods, and for each observer method:

- create an `Observer`, as defined in Section 10.6.1, “Observer method notification”, for the observer method and fire an event of type `ProcessObserverMethod`, as defined in Section 11.5.8, “ProcessObserverMethod event”.

The container determines which beans, interceptors and decorators are enabled, according to the rules defined in Section 2.5.5, “Enabled deployment types”, Section 9.4, “Interceptor enablement and ordering” and Section 8.2, “Decorator enablement and ordering”, taking into account any `<deploy>`, `<interceptors>` and `<decorators>` declarations in the `beans.xml` files, and registers the `Bean` and `Observer` objects:

- For each enabled bean that is not an interceptor or decorator, the container registers an instance of the `Bean` interface defined in Section 11.1, “The Bean interface”.
- For each enabled interceptor, the container registers an instance of the `Interceptor` interface defined in Section 11.1.2, “The Interceptor interface”.
- For each enabled decorator, the container registers an instance of the `Decorator` interface defined in Section 11.1.1, “The Decorator interface”.
- For each observer method of an enabled bean, the container registers an instance of the `Observer` interface defined in Section 10.2, “The Observer interface”.

12.4. 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 initialization time. If a definition error exists in a deployment, initialization will be aborted by the container.

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

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

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

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

```
public class InjectionException extends RuntimeException {

    public InjectionException() { super(); }
    public InjectionException(String message) { super(message); }
    public InjectionException(String message, Throwable cause) { super(message, cause); }
    public InjectionException(Throwable cause) { super(cause); }

}
```

This specification defines the following subclasses:

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

Appendix A. 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.

A.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.

A.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");
      }
      members = annotationType.getDeclaredMethods();
      for (Method member: members) {
         method.setAccessible(true);
      }
   }

   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 {
         return method.invoke(instance);
      }
      catch (IllegalArgumentException e) {
         throw new RuntimeException("Error checking value of member method " +
            method.getName() + " on " + method.getDeclaringClass(), e);
      }
      catch (IllegalAccessException e) {
         throw new RuntimeException("Error checking value of member method " +
            method.getName() + " on " + method.getDeclaringClass(), e);
      }
      catch (InvocationTargetException e) {
         throw new RuntimeException("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 B. Packages

The annotations and interfaces defined by this specification are divided into several packages in the `javax` namespace.

B.1. javax.interceptor

The following annotations are defined in the package `javax.interceptor`:

- Annotations: `@Interceptor`, `@InterceptorBindingType`

B.2. javax.decorator

The package `javax.decorator` contains annotations relating to decorators.

- Annotations: `@Decorator`, `@Decorates`

B.3. javax.enterprise.event

The package `javax.enterprise.event` contains annotations and interfaces relating to events.

- Annotation: `@Observes`
- Interfaces: `Event`, `Observer`
- Enumerations: `Notify`, `TransactionPhase`
- Exception: `ObserverException`

B.4. javax.enterprise.context

The package `javax.enterprise.context` contains annotations and interfaces relating to scopes and contexts.

- Annotation: `@ScopeType`
- Scope types: `@Dependent`, `@RequestScoped`, `@SessionScoped`, `@ApplicationScoped`, `@ConversationScoped`
- Interface: `Conversation`
- Exceptions: `ContextNotActiveException`, `BusyConversationException`, `NonexistentConversationException`

B.5. javax.enterprise.context.spi

The package `javax.enterprise.context.spi` contains the custom context SPI.

- Interfaces: `Context`, `Contextual`, `CreationalContext`

B.6. javax.enterprise.inject

The package `javax.enterprise.inject` contains annotations and interfaces relating to bindings, bean EL names, injection and programmatic lookup.

- Annotations: `@BindingType`, `@NonBinding`, `@Initializer`, `@Produces`, `@Disposes`, `@Named`, `@ConnectionFactory`
- Binding types: `@Any`, `@Current`, `@New`

- Interface: `Instance`
- Classes: `TypeLiteral`, `AnnotationLiteral`
- Exceptions: `InjectionException`, `UnsatisfiedResolutionException`, `AmbiguousResolutionException`, `UnproxyableResolutionException`, `CreationException`, `IllegalProductException`

B.7. javax.enterprise.inject.stereotype

The package `javax.enterprise.inject.stereotype` contains annotations relating to stereotypes.

- Annotation: `@Stereotype`
- Stereotype: `@Model`

B.8. javax.enterprise.inject.deployment

The package `javax.enterprise.inject.deployment` contains annotations relating to deployment types and specialization.

- Annotations: `@DeploymentType`, `@Specializes`
- Deployment types: `@Production`, `@Standard`

B.9. javax.enterprise.inject.spi

The package `javax.enterprise.inject.spi` contains the portable extension integration SPI.

- Interfaces: `BeanManager`, `InjectionPoint`, `Bean`, `Interceptor`, `Decorator`, `BeanClass`, `ManagedBean`, `SessionBean`, `ProducerBean`, `ObserverMethod`, `InjectionTarget`, `Producer`, `Listener`, `Extension`, `Annotated`, `AnnotatedType`, `AnnotatedMember`, `AnnotatedCallable`, `AnnotatedField`, `AnnotatedMethod`, `AnnotatedConstructor`, `AnnotatedParameter`, `BeforeBeanDiscovery`, `AfterBeanDiscovery`, `AfterDeploymentValidation`, `ProcessAnnotatedType`, `ProcessInjectionTarget`, `ProcessBean`, `ProcessManagedBean`, `ProcessSessionBean`, `ProcessProducerBean`, `ProcessObserverMethod`, `BeforeShutdown`
- Enumerations: `InterceptionType`, `SessionBeanType`