JAKARTA EE

Jakarta Contexts and Dependency Injection

Jakarta Contexts and Dependency Injection Specification Project

4.1, February 29, 2024: Draft(M1)

Table of Contents

Preface

Evaluation license

Organisation of this document

This document is organized in 4 parts:

- An introduction (this part), which is not part of the specification but introduces CDI concepts and gives examples.
- Core CDI specification: Part I - Core CDI. This part has two subparts:
  - CDI Lite specification: Part I.A - CDI Lite;
  - CDI Full specification: Part I.B - CDI Full.
- Specific CDI features for Java SE: Part II - CDI in Java SE.

Major changes

Jakarta Contexts and Dependency Injection 4.1

CDI 4.1 no longer specifies integration with Jakarta EE. This will now be specified in the Jakarta EE Platform, Web Profile and Core Profile specifications.

The Unified EL integration API in `BeanManager` has been deprecated and the relevant methods added to a dedicated interface `ELAwareBeanManager`, which is present in a new supplemental API artifact: `jakarta.enterprise:jakarta.enterprise.cdi-el-api`. This supplemental artifact declares a JPMS module `jakarta.cdi.el`, which declares a dependency on `jakarta.cdi` and `jakarta.el`.

The method invokers API has been added to allow frameworks to more easily call methods on managed beans and optionally look up the bean instance and arguments using CDI.

New methods have been added to `BeanContainer` to check assignability of beans and events and to obtain contexts for a scope.

The `@Priority` annotation can now be placed on producer methods and producer fields directly.

Jakarta Contexts and Dependency Injection 4.0

CDI 4.0 splits the CDI core into Lite and Full. Lite contains a subset of original features which are designed to work in more restricted environments. CDI Full contains everything that is in Lite plus all other features that were formerly in core CDI and are not in Lite.

A new `beans.xml` 4.0 schema file has been added and the namespace of the `beans_4_0.xsd` schema file is `xmlns:jakartaee="https://jakarta.ee/xml/ns/jakartaee"`, the same as 3.0. The key changes in the schema are to make the `bean-discovery-mode` attribute default to `annotated` and to use `annotated` as the default when an empty beans.xml is seen in a deployment. When running in a CDI Lite environment, only the `bean-discovery-mode attribute` is read from the beans.xml file. This means that by default, only those types with Bean defining annotations will be considered. Deployment relying on discovery of all types will either need to include a beans.xml with `bean-discovery-mode=all`, or introduce bean defining annotations to beans in the deployment.

A new jakarta.enterprise:jakarta.enterprise.lang-model API artifact has been aded for the Build Compatible (Reflection-Free) Java Language Model for CDI introduced by CDI Lite, and used in the new jakarta.enterprise.inject.build.compatible.spi package of the main jakarta.enterprise:jakarta.enterprise.cdi-api artifact. This new package is described in the Build compatible extensions section.

Java Platform Module System(JPMS) module-info.class files have been added to the CDI API artifacts. The cdi-api artifact defines a `jakarta.cdi` module with the following descriptor:

```
module jakarta.cdi {
    exports jakarta.decorator;
    exports jakarta.enterprise.context;
    exports jakarta.enterprise.context.control;
    exports jakarta.enterprise.context.spi;
    exports jakarta.enterprise.event;
    exports jakarta.enterprise.inject;
    exports jakarta.enterprise.inject.build.compatible.spi;
```

```
    exports jakarta.enterprise.inject.literal;
    exports jakarta.enterprise.inject.se;
    exports jakarta.enterprise.inject.spi;
    exports jakarta.enterprise.inject.spi.configurator;
    exports jakarta.enterprise.util;

    requires transitive jakarta.annotation;
    requires transitive jakarta.interceptor;
    requires transitive jakarta.cdi.lang.model;
    requires transitive jakarta.inject;
    requires static jakarta.el;
    // For javadoc
    requires static java.naming;
    //TODO: requires static jakarta.transation;

    uses jakarta.enterprise.inject.se.SeContainerInitializer;
    uses jakarta.enterprise.inject.spi.CDIProvider;
    uses jakarta.enterprise.inject.build.compatible.spi.BuildServices;
}
```

The lang-model artifact defines a `jakarta.cdi.lang.model` module with the following descriptor:

```
module jakarta.cdi.lang.model {
    exports jakarta.enterprise.lang.model;
    exports jakarta.enterprise.lang.model.declarations;
    exports jakarta.enterprise.lang.model.types;
}
```

Jakarta Contexts and Dependency Injection 3.0

CDI 3.0 is an incompatible update to Jakarta Contexts and Dependency Injection 2.0 (CDI 2.0).

CDI 3.0 includes a change in the base namespace used by the APIs from `javax` to `jakarta`. For example, the `BeanManager` interface has moved from `javax.enterprise.inject.spi.BeanManager` to `jakarta.enterprise.inject.spi.BeanManager`.

A new `beans.xml` 3.0 schema file was added and the namespace of the `beans_3_0.xsd` schema file has changed from `xmlns:javaee="http://xmlns.jcp.org/xml/ns/javaee"` to `xmlns:jakartaee="https://jakarta.ee/xml/ns/jakartaee"`.

Introduction

Chapter 1. Architecture

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

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

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

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

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

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 the Jakarta EE environment, the Java SE environment, or in other environments that implement CDI Lite. If the application uses Jakarta EE services such as transaction management and persistence in the Java SE environment, the services are usually restricted to, at most, the subset defined for embedded usage by the Jakarta Enterprise Bean 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 Jakarta EE container or an embeddable Jakarta Enterprise Bean container.

Concepts, Programming model, Inheritance, Interceptor bindings and Observer methods define the programming model for application 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. CDI Full adds Specialization and Decorator beans.

Dependency injection and lookup, Scopes and contexts, Lifecycle of contextual instances, Events and 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. CDI Full adds Decorators.

Packaging and deployment and Packaging and deployment in CDI Full defines how applications that use the services defined by this specification must be packaged into bean archives, and the responsibilities of the container implementation at application initialization time.

Build compatible extensions, Portable extensions, The `Contextual` interface and The `Context` interface define an SPI that allows portable extensions to integrate with the container.

1.2. Relationship to other specifications

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

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

1.2.1. Relationship to the Jakarta EE platform specification

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

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

qualifiers.

1.2.2. Relationship to Jakarta Enterprise Bean

Jakarta Enterprise Bean defines a programming model for application components that access transactional resources in a multi-user environment. Jakarta Enterprise Bean 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 Jakarta Enterprise Bean container at runtime.

Jakarta Enterprise Bean 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 Jakarta Enterprise Bean component model with contextual lifecycle management.

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

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

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

1.2.3. Relationship to Jakarta Dependency Injection

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

1.2.4. Relationship to Jakarta Interceptors

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

1.2.5. Relationship to Jakarta Faces

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

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

1.2.6. Relationship to Jakarta Validation

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

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

1.3. Introductory examples

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

1.3.1. Jakarta Faces example

The following Jakarta Faces 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 Jakarta EL expressions in this page refer to beans named `credentials` and `login`.

The `Credentials` bean has a lifecycle that is bound to the Jakarta Faces 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 Built-in stereotypes is a *stereotype* that identifies the `Credentials` bean as a model object in an MVC architecture.

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

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

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

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

    private User user;

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

    public void login() {

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

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

    }

    public void logout() {
        user = null;
    }

    public boolean isLoggedIn() {
```

```
        return user!=null;
    }

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

}
```

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

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

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

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

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

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

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

```
@Model
public class DocumentEditor {

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

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

}
```

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

When the login form is submitted, Jakarta Faces assigns the entered username and password to an instance of the `Credentials` bean that is automatically instantiated by the container. Next, Jakarta Faces 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. Jakarta Enterprise Bean example

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

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

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

    ...

    private User user;

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

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

    public void logout() {
        user = null;
    }

    public boolean isLoggedIn() {
```

```
        return user!=null;
    }

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

}
```

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

1.3.3. Jakarta EE component environment example

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

```
public class Databases {

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

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

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

}
```

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

1.3.4. Event example

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

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

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

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

    ...

    private User user;

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

    public void login() {

        List<User> results = ... ;

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

    }

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

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

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

}
```

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

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

```
@SessionScoped
public class Permissions implements Serializable {

    @Produces
```

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

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

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

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

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


}
```

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

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

1.3.5. Injection point metadata example

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

```
class Loggers {

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

```
}
```

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

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

```
@SessionScoped
public class Permissions implements Serializable {

    @Inject Logger log;

    ...

}
```

1.3.6. Interceptor example

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

The `AuthorizationInterceptor` class defines a custom authorization check:

```
@Secure @Interceptor
public class AuthorizationInterceptor {

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

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

```
}
```

The `@Interceptor` annotation, defined in 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 Interceptor binding types.

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

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

```
@Model
public class DocumentEditor {

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

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

}
```

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

1.3.7. Decorator example

NOTE Decorators are only available in CDI Full.

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

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

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

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

```
    public void delete(T object);

    public Class<T> getDataType();

}
```

The `DataAccessAuthorizationDecorator` class defines the authorization checks:

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

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

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

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

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

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

}
```

The `@Decorator` annotation defined in Declaring a decorator identifies the `DataAccessAuthorizationDecorator` class as a decorator. The `@Delegate` annotation defined in 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.

Part I - Core CDI

Structure

The Core CDI specification has two subparts:

- CDI Lite specification which contains a subset of CDI features and which can be implemented in more restricted environments; this is part of the Jakarta EE Core Profile;
- CDI Full specification that builds on top of Lite and adds all advanced CDI features; this is the classic CDI platform that is part of the Jakarta EE Web Profile and Jakarta EE Platform.

Part I.A - CDI Lite

Chapter 2. Concepts

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. A bean may bear metadata defining its lifecycle and interactions with other beans.

A bean comprises the following attributes:

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

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

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 Programming model.

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

The bean developer may also create interceptors or reuse existing interceptors. The interceptor bindings of a bean determine which interceptors will be applied at runtime. Interceptors are defined by Jakarta EE interceptors specification, and interceptor bindings are specified in Interceptor bindings.

In CDI Full environment, the bean developer may also create decorators or reuse existing decorators. The bean types and qualifiers of a bean determine which decorators will be applied at runtime. Decorators are defined in 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 Scopes and contexts and Lifecycle of contextual instances,
- scoped resolution by bean type and qualifier annotation type when injected into a Java-based client, as defined by Typesafe resolution,
- lifecycle callbacks and automatic injection of other bean instances, specified in Programming

model and Dependency injection and lookup,

- method interception, callback interception, as defined in Interceptor bindings, and
- event notification, as defined in Events.

In CDI Full environment, the container also provides the following capabilities:

- decoration, as defined in Decorators.

2.2. Bean types

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

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

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

The rules for determining the (unrestricted) set of bean types for a bean are defined in Bean types of a managed bean, Bean types of a producer method and Bean types of a producer field.

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 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, may be declared sealed or non-sealed or final, and may 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.

However, some Java types are not legal bean types :

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

- An array type whose component type is not a legal bean type.

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

2.2.2. Restricting the bean types of a bean

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

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

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

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

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

2.2.3. Typecasting between bean types

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

```
@Inject Business biz;
```

Then the following typecast is legal:

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

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

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

2.3. Qualifiers

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

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

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

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

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

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

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

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

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

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

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

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

An injection point may even specify multiple qualifiers.

Qualifier types are also used as event selectors by event consumers, as defined in Events.

In CDI Full environment, qualifier types are also used to bind decorators to beans, as specified in Decorators.

2.3.1. Built-in qualifier types

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

Every bean has the built-in qualifier `@Any`, even if it does not explicitly declare this qualifier.

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

The following declarations are equivalent:

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

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

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

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

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

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

```
public class Order {
    @Inject
```

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

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

2.3.2. Defining new qualifier types

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

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

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

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

A qualifier type may define annotation members.

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

2.3.3. Declaring the qualifiers of a bean

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

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

```
public class Shop {

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

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

}
```

Any bean may declare multiple qualifier types.

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

2.3.4. Specifying qualifiers of an injected field

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

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

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

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

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

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

2.3.5. Specifying qualifiers of a method or constructor parameter

Qualifier types may be applied to parameters of producer methods, initializer methods, disposer methods, observer methods or bean constructors (see 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 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;
}
```

2.3.6. Repeating qualifiers

In some cases, it may be useful to have a repeated qualifier for your type safe resolution. A repeated qualifier behaves just as any other qualifier does. For example, the below qualifier is a repeatable qualifier

```
@Target({ PARAMETER, FIELD, METHOD, TYPE })
@Retention(RUNTIME)
@Documented
@Qualifier
@Repeatable(Locations.class)
public @interface Location {
    String value();
}

@Target({ PARAMETER, FIELD, METHOD, TYPE })
@Retention(RUNTIME)
public @interface Locations {
    Location[] value();
}
```

Now you can define appropriate producers and injection points for repeated qualifiers.

```
@Produces
@Location("north")
@Location("south")
public Coordinate createCoordinate() {
   // ...
}

@Inject
@Location("north")
@Location("south")
private Coordinate coordinate;
```

A partial match injection point will still work in this case (from the same producer method)

```
@Inject
@Location("south")
private Coordinate coordinate;
```

However, adding the follow producer method will continue to give you an ambiguous resolution error (assuming the other producer exists as well)

```
@Produces
@Location("south")
public Coordinate createSouthCoordinate() {
   // ...
}
```

2.4. Scopes

Scoped objects, 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 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 { ... }
```

NOTE Session scope and conversation scope are only available in CDI Full.

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 three standard scope types defined in CDI Lite, all defined in the package `jakarta.enterprise.context`.

- The container must provide an implementation of the `@RequestScoped` and `@ApplicationScoped` annotations defined in Context management for built-in scopes. Note that these standard scopes can be extended by third-party extensions as defined in The `Context` interface
- Finally, there is a `@Dependent` pseudo-scope for dependent objects, as defined in Dependent pseudo-scope.

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

2.4.2. Defining new scope types

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

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

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

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

Custom scopes are normally defined by extensions, which must also provide an implementation of the `Context` interface, as defined in The `Context` interface, that implements the custom scope. Portable extensions provide a *context object* directly, while build compatible extensions provide a class that the container has to instantiate to obtain the context object.

2.4.3. Declaring the bean scope

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

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

```
public class Shop {

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

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

```
}
```

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

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

Alternatively, a scope type may be specified using a stereotype annotation, as defined in 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 present on the bean, directly, indirectly, or transitively, that declare different default scopes, then there is no default scope and the bean must explicitly declare a scope. If it does not explicitly declare a scope, the container automatically detects the problem and treats it as a definition error.

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

2.5. Default bean discovery mode

The default *bean discovery mode* for a bean archive is `annotated`, and such a bean archive is said to be an *implicit bean archive* as defined in Bean archives.

If the *bean discovery mode* is `annotated` then:

- bean classes that don't have *bean defining annotation* (as defined in Bean defining annotations) are not discovered, and
- producer methods (as defined in Producer methods) whose bean class does not have a *bean defining annotation* are not discovered, and
- producer fields (as defined in Producer fields) whose bean class does not have a *bean defining annotation* are not discovered, and
- disposer methods (as defined in Disposer methods) whose bean class does not have a *bean defining annotation* are not discovered, and

- observer methods (as defined in Declaring an observer method) whose bean class does not have a *bean defining annotation* are not discovered.

2.5.1. Bean defining annotations

A bean class may have a *bean defining annotation*, allowing it to be placed anywhere in an application, as defined in Bean archives. A bean class with a *bean defining annotation* is said to be an *implicit bean*.

The set of bean defining annotations contains:

- `@ApplicationScoped` and `@RequestScoped` annotations,
- all other normal scope types,
- `@Interceptor` annotation,
- all stereotype annotations (i.e. annotations annotated with `@Stereotype`),
- and the `@Dependent` scope annotation.

If one of these annotations is declared on a bean class, then the bean class is said to have a bean defining annotation. For example, this dependent scoped bean has a bean defining annotation:

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

whilst this dependent scoped bean does not have a bean defining annotation:

```
public class CoffeeShop
        extends Business
        implements Shop<Coffee> {
    ...
}
```

Note that to ensure compatibility with other Jakarta Dependency Injection implementations, all pseudo-scope annotations except `@Dependent` **are not** bean defining annotations. However, a stereotype annotation including a pseudo-scope annotation **is** a bean defining annotation.

2.6. Bean names

A bean may have a *bean name*. A bean with a name may be referred to by its name when used in a non typesafe environment (like the Unified Expression Language). A valid bean name is a period-separated list of valid EL identifiers.

The following strings are valid bean names:

```
com.acme.settings
```

```
orderManager
```

Subject to the restrictions defined in Ambiguous names, multiple beans may share the same bean name.

Bean names are used by the rules of bean name resolution defined in Name resolution.

2.6.1. Declaring the bean name

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

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

2.6.2. Default bean names

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

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

The default name for a bean depends upon the kind of the bean. The rules for determining the default name for a bean are defined in Default bean name for a managed bean, Default bean name for a producer method and Default bean name for a producer field.

2.6.3. Beans with no name

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

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

2.7. Alternatives

An *alternative* is a bean that must be explicitly selected if it should be available for lookup, injection or name resolution.

2.7.1. Declaring an alternative

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

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

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

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

2.8. Stereotypes

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

A stereotype encapsulates any combination of:

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

A stereotype may also specify that:

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

A bean may declare zero, one or multiple stereotypes.

2.8.1. Defining new stereotypes

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

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

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

2.8.1.1. Declaring the default scope for a stereotype

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

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

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

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

2.8.1.2. Specifying interceptor bindings for a stereotype

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

We may specify interceptor bindings that apply to all actions:

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

2.8.1.3. Declaring a `@Named` stereotype

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

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

We may specify that all actions have bean names:

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

A stereotype should not declare any qualifier annotation other than `@Named`. If a stereotype declares

any other qualifier annotation, non-portable behavior results.

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

2.8.1.4. Declaring an `@Alternative` stereotype

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

We may specify that all mock objects are alternatives:

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

2.8.1.5. Declaring stereotype with `@Priority`

A stereotype may declare a `@Priority` annotation which functions as a means of enabling and ordering affected beans.

If there are two different stereotypes present on a bean, directly, indirectly, or transitively, that declare different priority values, then the bean must explicitly declare a `@Priority` annotation. If the bean does not explicitly declare priority, the container automatically detects the problem and treats it as a definition error.

If a bean explicitly declares priority, any priority values declared by stereotypes are ignored.

Following sample shows a stereotype that can be used to mark bean as globally enabled alternative:

```
@Alternative
@Priority(Interceptor.Priority.APPLICATION + 5)
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {}
```

2.8.1.6. 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})`.

2.8.2. Declaring the stereotypes for a bean

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

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

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

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

Multiple stereotypes may be applied to the same bean:

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

2.8.3. Built-in stereotypes

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

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

In addition, the special-purpose `@Interceptor` stereotype is defined in Declaring the interceptor bindings of an interceptor.

2.9. Problems detected automatically by the container

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

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

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

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

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

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

Chapter 3. Programming model

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

- Managed beans
- Producer methods and fields

All containers must support managed beans, producer methods and producer fields.

Portable extensions and build compatible extensions may provide other kinds of beans.

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.

If a managed bean has a non-static public field, its scope must be a pseudo-scope (for example, `@Dependent` or `@Singleton`). If a managed bean with a non-static public field declares a normal scope, the container automatically detects the problem and treats it as a definition error.

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

3.1.1. Which Java classes are managed beans?

A Java class is a managed bean if it meets all of the following conditions:

- It is not an inner class.
- It is a non-abstract class.
- It does not implement `jakarta.enterprise.inject.spi.Extension` or `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension`.
- It is not annotated `@Vetoed` or in a package annotated `@Vetoed`.
- It has an appropriate constructor - either:
  - the class has a constructor with no parameters, or
  - the class declares a constructor annotated `@Inject`.

In CDI Full environment, a Java class is a managed bean also if:

- It is an abstract or non-abstract class annotated `@Decorator`.

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

If packages annotated `@Vetoed` are split across classpath entries, non-portable behavior results. An application can prevent packages being split across jars by sealing the package as defined by the

JAR File Specification.

3.1.2. Bean types of a managed bean

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

The resulting set of bean types for a managed bean consists only of legal bean types, all other types are removed from the set of bean types.

Note the additional restrictions upon bean types of beans with normal scopes defined in Unproxyable bean types.

3.1.3. Declaring a managed bean

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

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

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

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

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

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

A managed bean may extend another managed bean:

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

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

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

3.1.4. Default bean name for a managed bean

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

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

3.2. Producer methods

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

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

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

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 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 or is an array type whose component type contains a wildcard type parameter, the container automatically detects the problem and treats it as a definition error.

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

If a producer method return type is a type variable or an array type whose component type is a type variable, the container automatically detects the problem and treats it as a definition error.

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

A bean may declare multiple producer methods.

3.2.1. Bean types of a producer method

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

- If the return type is an interface, the unrestricted set of bean types contains the return type, all

interfaces it extends directly or indirectly and `java.lang.Object`.

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

The resulting set of bean types for a producer method consists only of legal bean types, all other types are removed from the set of bean types.

Note the additional restrictions upon bean types of beans with normal scopes defined in Unproxyable bean types.

3.2.2. Declaring a producer method

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

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

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

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

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

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

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

```
public class OrderFactory {

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

```
}
```

3.2.3. Default bean name for a producer method

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

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

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

This producer method is named `paymentProcessor`:

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

3.3. Producer fields

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

A producer field must be a default-access, public, protected or private, field of a managed bean class. 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 field contains a null value at runtime, and the producer field declares any other scope, an `IllegalProductException` is thrown by the container. This restriction allows the container to use a client proxy, as defined in 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 or is an array type whose component type contains a wildcard parameter, the container automatically detects the problem and treats it as a definition error.

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

If a producer field type is a type variable or is an array type whose component type is a type variable, the container automatically detects the problem and treats it as a definition error.

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

container.

A bean may declare multiple producer fields.

3.3.1. Bean types of a producer field

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

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

The resulting set of bean types for a producer field consists only of legal bean types, all other types are removed from the set of bean types.

Note the additional restrictions upon bean types of beans with normal scopes defined in Unproxyable bean types.

3.3.2. Declaring a producer field

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

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

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

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

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

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

3.3.3. Default bean name for a producer field

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

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

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

3.4. Disposer methods

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

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

A bean may declare multiple disposer methods.

3.4.1. Disposed parameter of a disposer method

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

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

3.4.2. Declaring a disposer method

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

```
public class UserDatabaseEntityManager {

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

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

}
```

```
public class Resources {

    private EntityManagerFactory emf;

    @PostConstruct
    public void setupEntityManagerFactory() {
        emf = Persistence.createEntityManagerFactory("userDatabase");
    }

    @Produces @UserDatabase
    public EntityManager start() {
        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.

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

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

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

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

3.4.3. Disposer method resolution

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

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

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

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

3.5. Bean constructors

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

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

3.5.1. Declaring a bean constructor

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

```
@SessionScoped
public class ShoppingCart implements Serializable {

   private User customer;

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

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

   ShoppingCart() {}

   ...

}
```

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

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

```
    }

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

    Order() {}

    ...

}
```

NOTE Session scope and conversation scope are only available in CDI Full.

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

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

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

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

3.6. Injected fields

An *injected field* is a non-static, non-final field of a bean class or of any other classes supporting injection.

3.6.1. Declaring an injected field

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

```
@ConversationScoped
public class Order {

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

}
```

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

3.7. Initializer methods

An *initializer method* is a default-access, public, protected or private, non-abstract, non-static, non-generic method of a bean class or of any other classes supporting injection.

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

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

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

3.7.1. Declaring an initializer method

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

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

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

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

}
```

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

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

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

3.8. The default qualifier at injection points

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

The following are equivalent:

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

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

}
```

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

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

}
```

The following definitions are equivalent:

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

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

}
```

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

   @Inject Payment(@Default Order order) {
```

```
        this(order.getAmount();
    }

}
```

Finally, the following are equivalent:

```
@Inject Order order;
```

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

3.9. The qualifier `@Named` at injection points

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

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

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

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

3.10. Unproxyable bean types

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

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

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

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

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

Chapter 4. Inheritance

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

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

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

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

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

The two cases are quite dissimilar.

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

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

Specialization is only present in CDI Full and is specified therein.

4.1. Inheritance of type-level metadata

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

- If X is annotated with a qualifier type, stereotype or interceptor binding type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares an annotation of type Z. (This behavior is defined by the Java Language Specification.)
- If X is annotated with a scope type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares a scope type. (This behavior is different to what is defined in the Java Language Specification.)

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

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

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

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

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

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

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

4.2. Inheritance of member-level metadata

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

- If X declares an injected field `x` then Y inherits `x`. (This behavior is defined by the Common Annotations for the Java Platform specification.)
- If X declares an initializer, non-static observer, `@PostConstruct` or `@PreDestroy` method `x()` then Y inherits `x()` if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y overrides the method `x()`. (This behavior is defined by the Common Annotations for the Java Platform specification.)
- If X declares a non-static method `x()` annotated with an interceptor binding type Z then Y inherits the binding if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y overrides the method `x()`. (This behavior is defined by the Common Annotations for the Java Platform specification.)
- If X declares a non-static producer or disposer method `x()` then Y does not inherit this method. (This behavior is different to what is defined in the Common Annotations for the Java Platform specification.)
- If X declares a non-static producer field `x` then Y does not inherit this field. (This behavior is different to what is defined in the Common Annotations for the Java Platform specification.)

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

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

```
public class DaoClient<T> {

    @Inject Dao<T> dao;
```

```
    ...

}
```

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

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

Chapter 5. Dependency injection and lookup

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, bean 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.

In general, a bean type or bean name does not uniquely identify a bean. When resolving a bean at an injection point, the container considers bean type, qualifiers and selected alternatives. 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 Normal scopes and pseudo-scopes. The container is not required to support circular chains of dependencies where every bean participating in the chain has a pseudo-scope.

5.1. Modularity

Beans and their clients may be deployed in *modules* in a module architecture. In a module architecture, certain modules are considered *bean archives*. In CDI Lite, a library that is a bean archive is always an implicit bean archive, as defined in Bean archives. Other kinds of bean archives exist in CDI Full.

A bean packaged in a certain module is available for injection, lookup and name resolution to classes packaged in some other module if and only if the bean class of the bean is required to be *accessible* to the other module by the class accessibility requirements of the module architecture.

An alternative is not available for injection, lookup or name resolution to classes in a module unless the module is a bean archive and the alternative is explicitly *selected* for the application.

5.1.1. Declaring selected alternatives

CDI Lite defines one method of selecting alternatives: the `@Priority` annotation allows an alternative to be selected for an entire application. CDI Full defines an additional method of selecting alternatives, which is specified therein.

5.1.1.1. Declaring selected alternatives for an application

An alternative may be given a priority for the application:

- by placing the `@Priority` annotation on the bean class of a managed bean, or
- by placing the `@Priority` annotation on the producer method, field or resource, or
- by placing the `@Priority` annotation on the bean class that declares the producer method, field

or resource, or

- by placing the `@Priority` annotation on a stereotype that is applied to the bean class, producer method or producer field.

5.1.2. Enabled and disabled beans

A bean is said to be *enabled* if:

- it is deployed in a bean archive, and
- it is not a producer method or field of a disabled bean, and either
- it is not an alternative, or it is a selected alternative for the application.

Otherwise, the bean is said to be disabled.

5.1.3. Inter-module injection

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

- the bean is not an interceptor
- the bean is enabled,
- the bean is either not an alternative, or the bean is a selected alternative for the application, and
- the bean class is required to be accessible to classes in the module, according to the class accessibility requirements of the module architecture.

5.2. Typesafe resolution

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

5.2.1. Performing typesafe resolution

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

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

- The bean has a bean type that matches the required type. For this purpose, primitive types are considered to match their corresponding wrapper types in `java.lang` and array types are considered to match only if their element types are identical. Parameterized and raw types are considered to match if they are identical or if the bean type is *assignable* to the required type, as defined in Assignability of raw and parameterized types.
- The bean has all the required qualifiers. If no required qualifiers were explicitly specified, the container assumes the required qualifier `@Default`. A bean has a required qualifier if it has a

qualifier with (a) the same type and (b) the same annotation member value for each member which is not annotated `@jakarta.enterprise.util.Nonbinding`.

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

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

5.2.2. Unsatisfied and ambiguous dependencies

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

When an ambiguous dependency exists, the container attempts to resolve the ambiguity. The container eliminates all eligible beans that are not alternatives, except for producer methods and fields of beans that are alternatives. If:

- there is exactly one bean remaining, the container will select this bean, and the ambiguous dependency is called resolvable.
- all the beans left are alternatives with a priority, or producer methods or fields of beans that are alternatives with a priority, then the container will determine the highest priority value and eliminate all beans, except for:
  - alternatives with the highest priority value,
  - producer methods and fields of alternatives, where either the producer method or field has the highest priority value, or the declaring alternative has the highest priority value.

  If there is exactly one bean remaining, the container will select this bean, and the ambiguous dependency is called resolvable.

For the purpose of determining the priority of any producer method or field during ambiguity resolution, the priority of the producer method or field is considered first. If the producer method or field does not have a priority, the priority of the managed bean that declares the producer method or field is used.

The container must validate all injection points of all enabled beans, all observer methods and all disposer methods 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.

5.2.3. Legal injection point types

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

If an injection point type is a type variable, the container automatically detects the problem and

treats it as a definition error.

5.2.4. Assignability of raw and parameterized types

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

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

- the required type parameter and the bean type parameter are actual types with identical raw type, and, if the type is parameterized, the bean type parameter is assignable to the required type parameter according to these rules, or
- the required type parameter is a wildcard, the bean type parameter is an actual type and the actual type is assignable to the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the required type parameter is a wildcard, the bean type parameter is a type variable and the upper bound of the type variable is assignable to or assignable from the upper bound, if any, of the wildcard and assignable from the lower bound, if any, of the wildcard, or
- the required type parameter is an actual type, the bean type parameter is a type variable and the actual type is assignable 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 `@Default Dao<Order>`, `@Default Dao<User>`, `@Default Dao<?>`, `@Default Dao<? extends Persistent>` or `@Default Dao<X extends Persistent>` where `X` is a type variable.

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

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

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

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

5.2.5. Primitive types and null values

For the purposes of typesafe resolution and dependency injection, primitive types and their corresponding wrapper types in the package `java.lang` are considered identical and assignable. If

necessary, the container performs boxing or unboxing when it injects a value to a field or parameter of primitive or wrapper type.

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

5.2.6. Qualifier annotations with members

Qualifier types may have annotation members.

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

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

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

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

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

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

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

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

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

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

5.2.7. Multiple qualifiers

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

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

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

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

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

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

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

5.3. Name resolution

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

A name resolves to a bean if:

- the bean has the given bean name, and
- the bean is available for injection in the module where the name resolution is requested.

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

5.3.1. Ambiguous names

An *ambiguous name* exists when a name resolves to multiple beans. When an ambiguous name exists, the container attempts to resolve the ambiguity. The container eliminates all eligible beans that are not alternatives selected for the application, except for producer methods and fields of beans that are alternatives. If:

- there is exactly one bean remaining, the container will select this bean, and the ambiguous dependency is called *resolvable*.
- all the beans left are alternatives with a priority, then the container will determine the highest priority value, and eliminate all beans, except for producer methods and fields of beans that are alternatives with the highest priority value. If there is exactly one bean remaining, the container will select this bean, and the ambiguous dependency is called *resolvable*.

All unresolvable ambiguous names are detected by the container when the application is

initialized. Suppose two beans are both available for injection in a certain module, and either:

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

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

5.4. Client proxies

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

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

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

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

5.4.1. Client proxy invocation

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

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

If the scope is not active, as specified in 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 class supporting injection. The resulting instance may or may not be a *contextual instance* as defined by Contextual instance of a bean.

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

- contextual instances of managed beans.

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

- non-contextual instances of managed beans.

The container interacts with instances of beans or objects supporting injection by calling methods and getting and setting field values.

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

5.5.1. Injection using the bean constructor

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

5.5.2. Injection of fields and initializer methods

When the container creates a new instance of a managed bean, the container must:

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

The container must ensure that:

- Initializer methods declared by a class X in the type hierarchy of the bean are called after all injected fields declared by X or by superclasses of X have been initialized.
- Any `@PostConstruct` callback declared by a class X in the type hierarchy of the bean is called after all initializer methods declared by X or by superclasses of X have been called, after all injected fields declared by X or by superclasses of X have been initialized.

5.5.3. Destruction of dependent objects

When the container destroys an instance of a bean, the container destroys all dependent objects, as

defined in Destruction of objects with scope `@Dependent`, after the `@PreDestroy` callback completes.

5.5.4. Invocation of producer or disposer methods

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

- If the method is static, the container must invoke the method.
- Otherwise, if the method is non-static, the container must:
- Obtain a contextual instance of the bean which declares the method, as defined by Contextual instance of a bean.
- Invoke the method upon this instance, as a business method invocation, as defined in 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 Destruction of objects with scope `@Dependent`.

5.5.5. Access to producer field values

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

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

5.5.6. Invocation of observer methods

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

- If the observer method is static, the container must invoke the method.
- Otherwise, if the observer method is non-static, the container must:
- Obtain a contextual instance of the bean which declares the observer method according to Contextual instance of a bean. If this observer method is a conditional observer method, obtain the contextual instance that already exists, only if the scope of the bean that declares the observer method is currently active, without creating a new contextual instance.
- Invoke the observer method on the resulting instance, if any, as a business method invocation, as defined in 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 Destruction of objects with scope `@Dependent`.

5.5.7. Injection point metadata

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

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

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

- The `getBean()` method returns the `Bean` object representing the bean that defines the injection point. If the injection point does not belong to a bean, `getBean()` returns a null value. If the injection point represents a dynamically obtained instance, the `getBean()` method should return the `Bean` object representing the bean that defines the `Instance` injection point.
- The `getType()` and `getQualifiers()` methods return the required type and required qualifiers of the injection point. If the injection point represents a dynamically obtained instance, the `getType()` and `getQualifiers()` methods should return the required type (as defined by `Instance.select()`), and required qualifiers of the injection point including any additional required qualifiers (as defined by `Instance.select()`).
- The `getMember()` method returns the `Field` object in the case of field injection, the `Method` object in the case of method parameter injection, or the `Constructor` object in the case of constructor parameter injection. If the injection point represents a dynamically obtained instance, the `getMember()` method returns the `Field` object representing the field that defines the `Instance` injection point in the case of field injection, the `Method` object representing the method that defines the `Instance` injection point in the case of method parameter injection, or the `Constructor` object representing the constructor that defines the `Instance` injection point in the case of constructor parameter injection.
- The `getAnnotated()` method may, in CDI Lite environment, always return `null`. Behavior of this method in CDI Full is specified therein.
- The `isDelegate()` method may, in CDI Lite environment, always return `false`. Behavior of this method in CDI Full is specified therein.
- The `isTransient()` method returns `true` if the injection point is a transient field, and `false` otherwise. If the injection point represents a dynamically obtained instance then the `isTransient()` method returns `true` if the `Instance` injection point is a transient field, and `false` otherwise.

Occasionally, a bean with scope `@Dependent` needs to access metadata relating to the object into

which it is injected. For example, the following producer method creates injectable `Logger` s. The log category of a `Logger` depends upon the class of the object into which it is injected:

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

The container must provide a bean with scope `@Dependent`, bean type `InjectionPoint` and qualifier `@Default`, allowing dependent objects, as defined in Dependent objects, to obtain information about the injection point to which they belong.

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

If a disposer method has an injection point of type `InjectionPoint` and qualifier `Default`, the container automatically detects the problem and treats it as a definition error.

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

5.5.8. Bean metadata

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

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

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

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

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

These beans are passivation capable dependencies, as defined in Passivation capable dependencies.

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

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

The injection of bean metadata is restricted. If:

- the injection point is a field, an initializer method parameter or a bean constructor, with qualifier `@Default`, then the type parameter of the injected `Bean`, or `Interceptor` must be the same as the type declaring the injection point, or
- the injection point is a field, an initializer method parameter or a bean constructor of an interceptor, with qualifier `@Intercepted`, then the type parameter of the injected `Bean` must be an unbounded wildcard, or
- the injection point is a producer method parameter then the type parameter of the injected `Bean` must be the same as the producer method return type, or
- the injection point is a parameter of a disposer method then the container automatically detects the problem and treats it as a definition error.

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

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

    @Inject Bean<OrderProcessor> bean;

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

}
```

5.6. Programmatic lookup

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

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

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

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

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

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

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

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

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

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

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

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

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

5.6.1. The `Instance` interface

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

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

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

    Stream<T> stream();

    boolean isUnsatisfied();
    boolean isAmbiguous();
    boolean isResolvable();

    void destroy(T instance);

    Handle<T> getHandle();
    Iterable<Handle<T>> handles();
    Stream<Handle<T>> handlesStream();
```

```
}
```

For an injected `Instance`:

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

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

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

The `select()` method returns a child `Instance` for a given required type and additional required qualifiers. If no required type is given, the required type is the same as the parent. Rules specified at The default qualifier at injection points are applied before performing typesafe resolution.

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

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

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

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

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

The `get()` method must:

- Identify a bean that has the required type and required qualifiers and is eligible for injection into the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Performing typesafe resolution, resolving ambiguities according to 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 Contextual reference for a bean.

The `iterator()` method must:

- Identify the set of beans that have the required type and required qualifiers and are eligible for injection into the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Performing typesafe resolution, resolving ambiguities

according to Unsatisfied and ambiguous dependencies.

- If typesafe resolution results in an unsatisfied dependency, the set of resulting beans is empty. If typesafe resolution results in an ambiguous dependency and the set of candidate beans contains at least one alternative, the set of resulting beans contains all beans that were not eliminated during ambiguity resolution. If typesafe resolution results in an ambiguous dependency and the set of candidate beans contains no alternative, the set of resulting beans contains all candidate beans.
- Return an `Iterator`, that iterates over the set of contextual references for the resulting beans and required type, as defined in Contextual reference for a bean.

The `stream()` method must:

- Identify the set of beans that have the required type and required qualifiers and are eligible for injection into the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Performing typesafe resolution, resolving ambiguities according to Unsatisfied and ambiguous dependencies.
- If typesafe resolution results in an unsatisfied dependency, the set of resulting beans is empty. If typesafe resolution results in an ambiguous dependency and the set of candidate beans contains at least one alternative, the set of resulting beans contains all beans that were not eliminated during ambiguity resolution. If typesafe resolution results in an ambiguous dependency and the set of candidate beans contains no alternative, the set of resulting beans contains all candidate beans.
- Return a `Stream`, that can stream over the set of contextual references for the resulting beans and required type, as defined in Contextual reference for a bean.

The methods `isUnsatisfied()`, `isAmbiguous()` and `isResolvable()` must:

- Identify the set of beans that have the required type and required qualifiers and are eligible for injection into the class into which the parent `Instance` was injected, according to the rules of typesafe resolution, as defined in Performing typesafe resolution, resolving ambiguities according to Unsatisfied and ambiguous dependencies.
- The method `isUnsatisfied()` returns `true` if there is no bean found, or `false` otherwise.
- The method `isAmbiguous()` returns `true` if there is more than one bean found, or `false` otherwise.
- The method `isResolvable()` returns `true` if there is exactly one bean found, or `false` otherwise.

The method `destroy()` instructs the container to destroy the instance. The bean instance passed to `destroy()` should be a dependent scoped bean instance obtained from the same `Instance` object, or a client proxy for a normal scoped bean. Applications are encouraged to always call `destroy()` when they no longer require an instance obtained from `Instance`. All built-in normal scoped contexts support destroying bean instances. An `UnsupportedOperationException` is thrown if the active context object for the scope type of the bean does not support destroying bean instances.

The `getHandle()` method must:

- Return an initialized contextual reference `Handle<T>` for a bean that has the required type and qualifiers and is eligible for injection. The contextual reference must be resolved lazily, i.e.

when first needed. The `Handle` interface is described in a separate paragraph.

- Throw `UnsatisfiedResolutionException` if there is no bean with given type and qualifiers.
- Throw `AmbiguousResolutionException` if there is more than one bean given type and qualifiers.

The `handles()` method must:

- Allow iterating over contextual reference handles for all beans that would be produced by the `iterator()` (or `stream()`) method.
- Return stateless `Iterable`. Therefore, each `Iterable#iterator()` produces a new set of handles.

The `handlesStream()` is a `Stream` equivalent of the aforementioned `handles()` method.

5.6.1.1. The `Handle` interface

`Handle` is an interface representing a contextual reference handle. It is an abstraction allowing inspection of bean metadata via `Bean<?>` objects. Handles have to resolve their contextual references lazily, i.e. when their `get()` method is invoked. Last but not least, this interface can be used to destroy the contextual instance once not needed.

```
public interface Handle<T> extends AutoCloseable {

        T get();
        Bean<T> getBean();
        void destroy();
        void close();

}
```

The `get()` method returns a contextual reference object. The contextual reference is resolved lazily. Throws `IllegalStateException` if invoked on `Handle` that previously successfully destroyed its underlying contextual reference.

The `getBean()` method returns the `Bean` object representing metadata of the given contextual instance.

The `destroy()` method destroys the contextual instance and is a no-op if:

- called multiple times
- the producing `Instance` does not exist
- the handle does not hold a contextual reference, i.e. `get()` was never called

The rules for destroying instances are the same as with `Instance#destroy()`.

The `close()` method delegates to the aforementioned `destroy()` method.

5.6.2. The built-in `Instance`

The container must provide a built-in bean that:

- is eligible for injection to any injection point with required type `Instance<X>` or `Provider<X>`, for any legal bean type `X`,
- has any qualifiers
- has scope `@Dependent`,
- has no bean name, and
- has an implementation provided automatically by the container.

5.6.3. Using `AnnotationLiteral` and `TypeLiteral`

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

```
public PaymentProcessor getSynchronousPaymentProcessor(PaymentMethod paymentMethod) {

    class SynchronousQualifier extends AnnotationLiteral<Synchronous>
            implements Synchronous {}

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

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

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

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

5.6.4. Built-in annotation literals

The following built-in annotations define a `Literal` static nested class to support inline instantiation of the specific annotation type:

- `jakarta.enterprise.inject.Any`
- `jakarta.enterprise.inject.Default`

- `jakarta.enterprise.inject.Specializes`
- `jakarta.enterprise.inject.Vetoed`
- `jakarta.enterprise.util.Nonbinding`
- `jakarta.enterprise.context.Initialized`
- `jakarta.enterprise.context.Destroyed`
- `jakarta.enterprise.context.RequestScoped`
- `jakarta.enterprise.context.SessionScoped`
- `jakarta.enterprise.context.ApplicationScoped`
- `jakarta.enterprise.context.Dependent`
- `jakarta.enterprise.context.ConversationScoped`
- `jakarta.enterprise.inject.Alternative`
- `jakarta.enterprise.inject.Typed`

The `Literal` class might be used to instantiate the matching `AnnotationLiteral`:

```
Default defaultLiteral = new Default.Literal();
```

Annotations without members provide the default `AnnotationLiteral` instance declared as a constant named `INSTANCE`:

```
RequestScoped requestScopedLiteral = RequestScoped.Literal.INSTANCE;
```

Annotations having members do not provide the default `AnnotationLiteral` instance. Instead, a constructor or factory method named `of` can be used:

```
Initialized initializedForApplicationScoped = new Initialized.Literal
(ApplicationScoped.class);

Initialized initializedForRequestScoped = Initialized.Literal.of(RequestScoped.class);
```

See also the annotation javadoc for more information about specific `Literal` members.

In addition, CDI also provides annotation literals for the following JSR 330 annotations:

- `jakarta.inject.Inject` with `jakarta.enterprise.inject.literal.InjectLiteral` class
- `jakarta.inject.Named` with `jakarta.enterprise.inject.literal.NamedLiteral` class
- `jakarta.inject.Qualifier` with `jakarta.enterprise.inject.literal.QualifierLiteral` class
- `jakarta.inject.Singleton` with `jakarta.enterprise.inject.literal.SingletonLiteral` class

They can be used like static nested classes described above.

Chapter 6. Scopes and contexts

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

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

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

6.1. The `Contextual` interface

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

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

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

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

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

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

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

6.1.1. The `CreationalContext` interface

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

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

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

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

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

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

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

6.2. The `Context` interface

The `jakarta.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. It should not be called directly by the application.

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

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

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

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

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

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

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

The `AlterableContext` interface was introduced in CDI 1.1 to allow bean instances to be destroyed by the application. Extensions providing context implementations for normal scopes should implement `AlterableContext` instead of `Context`.

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

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

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

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

The `get()` operation of the context object for an active normal scope returns the current instance of the given contextual type.

At certain points in the execution of the program a context may be *destroyed*. When a context is destroyed, all mapped instances belonging to that context are destroyed by passing them to the `Contextual.destroy()` method.

Contexts with normal scopes must obey the following rule:

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

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

All normal scopes must be explicitly declared `@NormalScope`, to indicate to the container that a client proxy is required.

All pseudo-scopes must be explicitly declared `@Scope`, to indicate to the container that no client proxy is required.

All scopes defined by this specification, except for the `@Dependent` pseudo-scope, are normal scopes.

6.4. Dependent pseudo-scope

The `@Dependent` scope type is a pseudo-scope. Beans declared with scope type `@Dependent` behave differently to beans with other built-in scope types.

When a bean is declared to have `@Dependent` scope:

- No injected instance of the bean is ever shared between multiple injection points.
- Any instance of the bean injected into an object that is being created by the container is bound to the lifecycle of the newly created object.
- Any instance of the bean that receives a producer method, producer field, disposer method or observer method invocation exists to service that invocation only.
- Any instance of the bean injected into method parameters of a disposer method or observer method exists to service the method invocation only (except for observer methods of container lifecycle events).

Every invocation of the `get()` operation of the `Context` object for the `@Dependent` scope with a `CreationalContext` returns a new instance of the given bean.

Every invocation of the `get()` operation of the `Context` object for the `@Dependent` scope with no

`CreationalContext` returns a null value.

The `@Dependent` scope is always active.

6.4.1. Dependent objects

Many instances of beans with scope `@Dependent` belong to some other bean and are called *dependent objects*.

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

6.4.2. Destruction of objects with scope `@Dependent`

Dependent objects of a contextual instance are destroyed when `Contextual.destroy()` calls `CreationalContext.release()`, as defined in The `CreationalContext` interface.

Additionally, the container must ensure that:

- all dependent objects of a non-contextual instance of a bean are destroyed when the instance is destroyed by the container,
- all `@Dependent` scoped contextual instances injected into method parameters of a disposer method or an observer method are destroyed when the invocation completes,
- all `@Dependent` scoped contextual instances injected into method or constructor parameters that are annotated with `@TransientReference` are destroyed when the invocation completes, and
- any `@Dependent` scoped contextual instance created to receive a producer method, producer field, disposer method or observer method invocation is destroyed when the invocation completes.

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. Activating Built In Contexts

Certain built in contexts support the ability to be activated and deactivated. This allows developers to control built-in contexts in ways that they could also manage custom built contexts.

When activating and deactivating built in contexts, it is important to realize that they can only be activated if not already active within a given thread.

6.5.2.1. Activating a Request Context

Request contexts can be managed either programmatically or via interceptor.

To programmatically manage request contexts, the container provides a built in bean that is `@Dependent` scoped and of type `RequestContextController` that allows you to activate and deactivate a request context on the current thread. The object should be considered stateful, invoking the same instance on different threads may not work properly, non-portable behavior may occur.

```
public interface RequestContextController {
   boolean activate();
   void deactivate() throws ContextNotActiveException;
}
```

When the `activate()` method is called, if the request context is not already active on the current thread then it will be activated and the method returns `true`. Otherwise, the method returns `false`.

When the `deactivate()` method is called, if this controller started the request context then the request context is stopped. The method does nothing if this controller did not activate the context and the context is active. This method throws a `ContextNotActiveException` if there is no request context active.

If within the same thread the `activate()` and `deactivate()` methods are invoked repeatedly, the instances between the activations are not the same instances, each request context should be assumed to be new.

To automatically start a request context via interceptor binding, the container provides an interceptor `@ActivateRequestContext` which will activate a request context if not already active prior to the method's invocation, and deactivate it upon method completion, with the same rules as in `RequestContextController`. The interceptor is automatically registered with a priority of `PLATFORM_BEFORE + 100`.

6.5.3. Contextual instance of a bean

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

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

From time to time, the container attempts to obtain a *contextual instance of a bean that already exists*, without creating a new contextual instance. The container must determine if the scope of the bean is active and if it is:

- obtain the active context object for the bean scope, then
- attempt to obtain an existing instance of the bean by calling `Context.get()`, passing the `Bean` instance representing the bean without passing any instance of `CreationalContext`.

If the scope is not active, or if `Context.get()` returns a null value, there is no contextual instance that already exists.

A contextual instance of any of the built-in kinds of bean defined in 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.

6.5.4. Contextual reference for a bean

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

Contextual references must be obtained with a given `CreationalContext`, allowing any instance of scope `@Dependent` that is created to be later destroyed.

- If the bean has a normal scope and the given bean type cannot be proxied by the container, as defined in 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 Client proxies, created by the container, that implements the given bean type and all bean types of the bean which are Java interfaces.
- Otherwise, if the bean has a pseudo-scope, the container must obtain a contextual instance of the bean. If the bean has scope `@Dependent`, the container must associate it with the `CreationalContext`.

The container must ensure that every injection point of type `InjectionPoint` and qualifier `@Default` of any dependent object instantiated during this process receives:

- an instance of `InjectionPoint` representing the injection point into which the dependent object

will be injected, or

- a null value if it is not being injected into any injection point.

6.5.5. Contextual reference validity

A contextual reference for a bean is *valid* only for a certain period of time. The application should not invoke a method of an invalid reference.

The validity of a contextual reference for a bean depends upon whether the scope of the bean is a normal scope or a pseudo-scope.

- Any reference to a bean with a normal scope is valid as long as the application maintains a hard reference to it. However, it may only be invoked when the context associated with the normal scope is active. If it is invoked when the context is inactive, a `ContextNotActiveException` is thrown by the container.
- Any reference to a bean with a normal scope is invalid after CDI container shutdown. Maintaining such reference and attempting to use it after container shutdown results in an `IllegalStateException`.
- Any reference to a bean with a pseudo-scope (such as `@Dependent`) is valid until the bean instance to which it refers is destroyed. It may be invoked even if the context associated with the pseudo-scope is not active. If the application invokes a method of a reference to an instance that has already been destroyed, the behavior is undefined.

6.5.6. Injectable references

From time to time, the container must obtain an *injectable reference* for an injection point. The container must:

- Identify a bean according to the rules defined in Typesafe resolution and resolving ambiguities according to Unsatisfied and ambiguous dependencies.
- Obtain a contextual reference for this bean and the type of the injection point according to 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 Contextual instance of a bean.
- If an incompletely initialized instance of the bean is registered with the current `CreationalContext`, as defined in The `Contextual` interface, the container is permitted to directly inject this instance.

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

6.5.7. Injectable reference validity

Injectable references to a bean must respect the rules of contextual reference validity, with the

following exceptions:

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

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

6.6. Context management for built-in scopes

The container must provide an implementation of the `Context` interface for each of the built-in scopes defined in Built-in scope types. These implementations depend on the platform the container is running.

The context associated with a built-in normal scope propagates across local, synchronous Java method calls. The context does not propagate across remote method invocations or to asynchronous processes.

6.6.1. Request context lifecycle

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

An event with qualifier `@Initialized(RequestScoped.class)` is synchronously fired when the request context is initialized. An event with qualifier `@BeforeDestroyed(RequestScoped.class)` is synchronously fired when the request context is about to be destroyed, i.e. before the actual destruction. An event with qualifier `@Destroyed(RequestScoped.class)` is synchronously fired when the request context is destroyed, i.e. after the actual destruction.

The request context is active:

- during notification of an asynchronous observer method, and
- during `@PostConstruct` callback of any bean.

The request context is destroyed:

- after the invocation of an asynchronous observer method completes, and
- after the `@PostConstruct` callback completes, if it did not already exist when the `@PostConstruct` callback occurred.

6.6.2. Application context lifecycle

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

An event with qualifier `@Initialized(ApplicationScoped.class)` is synchronously fired when the application context is initialized. An event with qualifier `@BeforeDestroyed(ApplicationScoped.class)` is synchronously fired when the application context is about to be destroyed, i.e. before the actual destruction. An event with qualifier `@Destroyed(ApplicationScoped.class)` is synchronously fired when the application context is destroyed, i.e. after the actual destruction.

6.7. Context management for custom scopes

Custom context implementations are encouraged to synchronously fire:

- an event with qualifier `@Initialized(X.class)` when a custom context is initialized, i.e. ready for use,
- an event with qualifier `@BeforeDestroyed(X.class)` when a custom context is about to be destroyed, i.e. before the actual destruction,
- an event with qualifier `@Destroyed(X.class)` when a custom context is destroyed, i.e. after the actual destruction,

where `X` is the scope type associated with the context.

A suitable event payload should be chosen.

Build compatible extensions may define custom context classes for custom scopes, but they may not define custom context classes for built-in scopes.

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 Scopes and contexts.

Every bean in the system is represented by an instance of the `Bean` interface defined in 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 The `Contextual` interface.

7.1. Restriction upon bean instantiation

There are 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 Contextual instance of a bean. Furthermore, the capabilities listed in 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 name resolution, but the other capabilities listed in 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:

```
@RequestScoped
public class PaymentStrategyProducer implements Serializable {

   private PaymentStrategyType paymentStrategyType;

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

   @Produces PaymentStrategy getPaymentStrategy(@CreditCard PaymentStrategy
creditCard,
                                                @Cheque PaymentStrategy cheque,
                                                @Online PaymentStrategy online) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return creditCard;
```

```
            case CHEQUE: return cheque;
            case ONLINE: return online;
            default: throw new IllegalStateException();
        }
    }

}
```

In this case, any object returned by the producer method has already had its dependencies injected, receives lifecycle callbacks and event notifications and may have interceptors.

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

```
@RequestScoped
public class PaymentStrategyProducer implements Serializable {

   private PaymentStrategyType paymentStrategyType;

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

   @Produces PaymentStrategy getPaymentStrategy() {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return new CreditCardPaymentStrategy();
         case CHEQUE: return new ChequePaymentStrategy();
         case ONLINE: return new OnlinePaymentStrategy();
         default: throw new IllegalStateException();
      }
    }

}
```

In this case, any object returned by the producer method will not have any dependencies injected by the container, receives no lifecycle callbacks or event notifications and does not have interceptors or, in CDI Full, decorators.

7.2. Container invocations and interception

When the application invokes a method of a bean via a contextual reference to the bean, as defined in Contextual reference for a bean, the invocation is treated as a *business method invocation*.

When the container invokes a method of a bean, the invocation may or may not be treated as a business method invocation:

- Invocations of initializer methods by the container are not business method invocations.
- Invocations of producer, disposer and observer methods by the container are business method invocations and are intercepted by method interceptors.

- Invocation of lifecycle callbacks by the container are not business method invocations, but are intercepted by interceptors for lifecycle callbacks.
- Invocations of interceptor methods during method or lifecycle callback interception are not business method invocations, and therefore no recursive interception occurs.
- Invocations of methods declared by `java.lang.Object` are not business method invocations.

A method invocation passes through method interceptors if, and only if, it is a business method invocation.

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

7.3. Lifecycle of contextual instances

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

7.3.1. Lifecycle of managed beans

When the `create()` method of the `Bean` object that represents a managed bean is called, the container obtains an instance of the bean, calling the bean constructor as defined by Injection using the bean constructor, and performing dependency injection as defined in Injection of fields and initializer methods.

When the `destroy()` method is called, the container destroys the instance, and any dependent objects, as defined in Destruction of dependent objects.

7.3.2. 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 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 Invocation of producer or disposer methods, passing the instance given to `destroy()` to the disposed parameter. Finally, the container destroys dependent objects, as defined in Destruction of dependent objects.

7.3.3. 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 Access to producer field values to obtain the current value of the field. The value of the producer field is the new contextual instance to be returned by `Bean.create()`.

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

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

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

Chapter 8. Interceptor bindings

Managed beans support interception. *Interceptors* are used to separate cross-cutting concerns from business logic. The Jakarta Interceptors specification defines the basic programming model and semantics, and how to associate interceptors with target classes. This specification defines an extent to which CDI Lite supports Jakarta Interceptors specification, including extending it with non-binding annotation values in interceptor resolution.

CDI Lite implementations are required to support following forms of interception:

- Interceptors declared on interceptor classes and associated with target class using interceptor binding annotations
  - `@AroundInvoke`, `@PostConstruct`, `@PreDestroy` and `@AroundConstruct` are all supported
  - Enablement and ordering of interceptors using `@Priority` annotation
- `@PostConstruct` and `@PreDestroy` declared on target class (i.e. on a bean)

Using other forms of interception results in non-portable behavior.

CDI Full implementations are required to support all forms of interception, as described in Interceptor bindings in CDI Full.

8.1. Interceptor binding types

This specification extends the Jakarta Interceptors specification and allows interceptor bindings to be applied to CDI stereotypes.

8.1.1. Interceptor bindings for stereotypes

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

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

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

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

8.2. Declaring the interceptor bindings of an interceptor

This specification extends the Jakarta Interceptors specification and defines how the container must behave if a definition error is encountered.

If an interceptor declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

8.3. Binding an interceptor to a bean

This specification extends the Jakarta Interceptors specification and defines:

- additional restrictions about the type of bean to which an interceptor can be bound, and
- how the container must behave if a definition error is encountered, and
- how interceptors are bound using stereotypes.

Interceptor bindings may be used to associate interceptors with any managed bean.

The set of interceptor bindings for a method declared at class level includes those declared on stereotypes. An interceptor binding declared on a bean class replaces an interceptor binding of the same type declared by a stereotype that is applied to the bean class.

The set of interceptor bindings for a producer method is not used to associate interceptors with the return value of the producer method.

If a managed bean has a class-level or method-level interceptor binding, the managed bean must be a proxyable bean type, as defined in Unproxyable bean types.

8.4. Interceptor resolution

This specification extends the Jakarta Interceptors specification and defines the effect of applying `@Nonbinding` to an interceptor binding member.

If any interceptor binding has a member annotated `@jakarta.enterprise.util.Nonbinding`, the member is ignored when performing interceptor resolution, and the method does not need to have the same annotation member value.

Chapter 9. Events

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

An event comprises:

- A Java object - the *event object*
- A set of instances of qualifier types - the *event qualifiers*

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

An *observer method* acts as event consumer, observing events of a specific type - the *observed event type* - with a specific set of qualifiers - the *observed event qualifiers*. An observer method will be notified of an event if the event object is assignable to the observed event type, and if the set of observed event qualifiers is a subset of all the event qualifiers of the event.

9.1. Event types and qualifier types

An event object is an instance of a concrete Java class with no unresolvable 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 an unresolvable type variable. A wildcard type is not considered an unresolvable type variable.

An event qualifier type is just an ordinary qualifier type as specified in Defining new qualifier types, typically defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})` or `@Target({FIELD, PARAMETER})`.

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

Any Java type may be an observed event type.

9.2. Firing events

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

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

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

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

Or the application may specify qualifiers dynamically:

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

In this example, the event sometimes has the qualifier `@Admin`, depending upon the value of `user.isAdmin()`. It is first fired synchronously with `fire()` then asynchronously with `fireAsync()`.

9.2.1. Firing events synchronously

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

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

Event fired with the `fire()` method is fired synchronously. All the resolved synchronous observers (as defined in Observer resolution) are called in the thread in which `fire()` was called. A synchronous observer notification blocks the calling thread until it completes.

9.2.2. Firing events asynchronously

Events may also be fired asynchronously using one of the methods `fireAsync()`

```
@Inject Event<LoggedInEvent> loggedInEvent;

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

Event fired with the `fireAsync()` method is fired asynchronously. All the resolved asynchronous observers (as defined in Observer resolution) are called in one or more different threads.

Method `fireAsync()` returns immediately.

9.2.3. The Event interface

The Event interface provides a method for firing events with a specified combination of type and qualifiers:

```
public interface Event<T> {

    public void fire(T event);
    public <U extends T> CompletionStage<U> fireAsync(U event);
    public <U extends T> CompletionStage<U> fireAsync(U event, NotificationOptions
options);

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

}
```

For an injected Event:

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

For example, this injected Event has specified type LoggedInEvent:

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

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

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

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

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

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

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

The methods fire() and fireAsync() fire an event with the specified qualifiers and notify observers,

as defined by Observer notification. If the container is unable to resolve the parameterized type of the event object, it uses the specified type to infer the parameterized type of the event types.

The method `fireAsync()` may be called with a `NotificationOptions` object to configure the observer methods notification, e.g. to specify an `Executor` object to be used for asynchronous delivery. The container is permitted to define other non-portable notification options.

The following elements are container specific:

- the default `Executor` used by the container when `fireAsync()` is called without specifying an `Executor`,
- the `CompletionStage` returned by `fireAsync` methods, and
- all dependent stages of this initial `CompletionStage`.

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

If the runtime type of the event object is assignable to the type of a container lifecycle event, an `IllegalArgumentException` is thrown.

9.2.4. The built-in `Event`

The container must provide a built-in bean with:

- `Event<X>` in its set of bean types, for every Java type `X` that does not contain a type variable,
- every event qualifier type in its set of qualifier types,
- scope `@Dependent`,
- no bean name, and
- an implementation provided automatically by the container.

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

9.3. Observer resolution

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

Observer resolution usually occurs at runtime.

An event is delivered to an observer method if:

- The observer method belongs to an enabled bean.
- An event type is assignable to the observed event type, taking type parameters into consideration.
- The observer method has no event qualifiers or has a subset of the event qualifiers. An observer method has an event qualifier if it has an observed event qualifier with (a) the same type and

(b) the same annotation member value for each member which is not annotated `@jakarta.enterprise.util.Nonbinding`.

- Either the event is not a container lifecycle event, as defined in Container lifecycle events, or the observer method belongs to an extension.
- The event is fired synchronously and the observer is a synchronous observer as defined in Declaring an observer method.
- The event is fired asynchronously and the observer is an asynchronous observer as defined in Declaring an observer method.

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

9.3.1. Assignability of type variables, raw and parameterized types

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

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

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.

9.3.2. Event qualifier types with members

As usual, the qualifier type may have annotation members:

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

Consider the following event:

```
@Inject Event<LoggedInEvent> loggedInEvent;
...
public void login() {
    final User user = ...;
    loggedInEvent.select(new RoleQualifier() { public String value() { return user
.getRole(); } }).fire(new LoggedInEvent(user));
}
```

Where `RoleQualifier` is an implementation of the qualifier type `Role`:

```
public abstract class RoleQualifier
        extends AnnotationLiteral<Role>
        implements Role {}
```

Then the following observer method will always be notified of the event:

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

Whereas this observer method may or may not be notified, depending upon the value of `user.getRole()`:

```
public void afterAdminLogin(@Observes @Role("admin") LoggedInEvent event) { ... }
```

As usual, the container uses `equals()` to compare event qualifier type member values.

9.3.3. Multiple event qualifiers

An event parameter may have multiple qualifiers.

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

Then this observer method will be notified if the set of observer qualifiers is a subset of the fired event's qualifiers or an empty set:

```
@Inject Event<Document> documentEvent;
...
documentEvent.select(new UpdatedQualifier(), new ByAdminQualifier(), new
ClarificationQualifier()).fire(document);
```

In the above example the event is fired with `@ByAdmin`, `@Updated`, and `@Clarification` qualifiers. The

observer qualifiers are `@Updated` and `@ByAdmin`. Observer qualifiers therefore form a subset of event qualifiers and the observer will be notified.

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

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

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

On the other hand, following observer will not be notified as slightly different behaviour applies to observers with `@Default` qualifier:

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

Such observer will only be notified for events having either no qualifiers or only `@Default` qualifier:

```
@Inject Event<Document> documentEvent;
@Inject @Default Event<Document> documentDefaultEvent;
...
documentEvent.fire(document);
documentDefaultEvent.fire(document);
```

9.4. Observer methods

An observer method allows the application to receive and respond to event notifications.

An observer method is a non-abstract method of a managed bean class. An observer method may be either static or non-static.

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

A bean may declare multiple observer methods.

9.4.1. Event parameter of an observer method

Each observer method must have exactly one *event parameter*, of the same type as the event type it observes. When searching for observer methods for an event, the container considers the type and qualifiers of the event parameter.

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

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

The event parameter may be an array type whose component type contains a type variable or a wildcard.

Modifications made to the event parameter in an observer method are propagated to following observers. The container is not required to guarantee a consistent state for an event parameter modified by asynchronous observers.

9.4.2. Declaring an observer method

An observer method may be declared by annotating a parameter `@jakarta.enterprise.event.Observes` or `@jakarta.enterprise.event.ObservesAsync` of a default-access, public, protected or private method. That parameter is the event parameter. The declared type of the parameter is the observed event type.

If `@Observes` is used the observer method is a synchronous observer method.

If `@ObservesAsync` is used the observer method is an asynchronous observer method.

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

public void asyncAfterLogin(@ObservesAsync LoggedInEvent event) { ... }
```

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

If a method has a parameter annotated `@Observes` and `@ObservesAsync`, the container automatically detects the problem and treats it as a definition error.

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

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

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

Interceptors may not declare observer methods. If an interceptor has a method with a parameter annotated `@Observes` or `@ObservesAsync`, the container automatically detects the problem and treats it as a definition error.

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

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

9.4.3. The `EventMetadata` interface

The interface `jakarta.enterprise.inject.spi.EventMetadata` provides access to metadata about an observed event.

```
public interface EventMetadata {
    public Set<Annotation> getQualifiers();
    public InjectionPoint getInjectionPoint();
    public Type getType();
}
```

- `getQualifiers()` returns the set of qualifiers with which the event was fired.
- `getInjectionPoint()` returns the `InjectionPoint` from which this event payload was fired, or `null` if it was fired from `BeanContainer.getEvent()`.
- `getType()` returns the type representing runtime class of the event object with type variables resolved.

The container must provide a bean with scope `@Dependent`, bean type `EventMetadata` and qualifier `@Default`, allowing observer methods to obtain information about the events they observe.

If an injection point of type `EventMetadata` and qualifier `@Default` which is not a parameter of an observer method exists, the container automatically detects the problem and treats it as a definition error.

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

9.4.4. 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 `notifyObserver=IF_EXISTS`.

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

public void asyncRefreshOnDocumentUpdate(@ObservesAsync(notifyObserver=IF_EXISTS)
@Updated Document doc) { ... }
```

Beans with scope `@Dependent` may not have conditional observer methods. If a bean with scope `@Dependent` has an observer method declared `notifyObserver=IF_EXISTS`, the container automatically detects the problem and treats it as a definition error.

The enumeration `jakarta.enterprise.event.Reception` identifies the possible values of `notifyObserver`:

```
public enum Reception { IF_EXISTS, ALWAYS }
```

9.4.5. 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.

If the transaction is in progress, but `jakarta.transaction.Synchronization` callback cannot be registered due to the transaction being already marked for rollback or in state where `jakarta.transaction.Synchronization` callbacks cannot be registered, the *before completion*, *after completion* and *after failure* observer methods are notified at the same time as other observers, but *after_success* observer methods get skipped.

- 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 `jakarta.enterprise.event.TransactionPhase` identifies the kind of transactional observer method:

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

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

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

Asynchronous observer cannot be declared transactional.

9.5. Observer notification

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

- determine the observer methods for that event according to the rules of observer resolution defined by Observer resolution, then,
- for each observer method, either invoke the observer method immediately, or register the observer method for later invocation during the transaction completion phase, using a JTA `Synchronization`.
- honor the priority of observer methods as defined in Observer ordering.

The container calls observer methods as defined in Invocation of observer methods.

- If the observer method is a transactional observer method and there is currently a JTA transaction in progress, the container calls the observer method during the appropriate transaction completion phase.
- If there is no context active for the scope to which the bean declaring the observer method belongs, then the observer method should not be called.
- Otherwise, the container calls the observer immediately.

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

Observer methods may throw exceptions:

- If the observer method is a transactional observer method, any exception is caught and logged by the container.
- If the observer method is asynchronous, the exception aborts processing of the observer but not of the event. Exception management during an asynchronous event is defined in Handling exceptions thrown during an asynchronous event.
- Otherwise, the exception aborts processing of the event. No other observer methods of that event will be called. The `Event.fire()` method rethrows the exception. If the exception is a checked exception, it is wrapped and rethrown as an (unchecked) `ObserverException`.

9.5.1. Handling exceptions thrown during an asynchronous event

If an event is asynchronous, and an exception is thrown by one or more of its notified observers, the `CompletionStage` returned by `fireAsync` will complete exceptionally with `java.util.concurrent.CompletionException`. `CompletionException` contains all exceptions thrown by observers as suppressed exceptions. They can be accessed as an array of `Throwable` with the `getSuppressed` method.

It can be handled with one of the `CompletionStage` methods related to exceptions:

```
myEvent.fireAsync(anEventObject)
       .handle((ok, ex) -> {
           if (ok != null) {
               return ok;
           } else {
             for (Throwable t : ex.getSuppressed()) {
```

```
                    ...
            }
              ...
          } });
```

If no exception is thrown by observers then the resulting `CompletionStage` is completed normally with the event object.

9.5.2. Observer ordering

Before the actual observer notification, the container determines an order in which the observer methods for a certain event are invoked. The priority of an observer method may be declared by annotating the event parameter with `@Priority` annotation. If a `@Priority` annotation is declared on an event parameter of an asynchronous observer method, non-portable behavior results. If no `@Priority` annotation is specified, the default priority `jakarta.interceptor.Interceptor.Priority.APPLICATION + 500` is assumed. Observers with smaller priority values are called first.

```
void afterLogin(@Observes @Priority(jakarta.interceptor.Interceptor.Priority
.APPLICATION) LoggedInEvent event) { ... }
```

The order of more than one observer with the same priority is undefined and the observer methods are notified therefore in a non predictable order.

9.5.3. Observer method invocation context

The transaction 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 asynchronous, it is called in a new lifecycle contexts and a new transaction context. As specified in Context management for built-in scopes, contexts associated with built-in normal scope don't propagate across asynchronous observers.
- 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 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 lifecycle contexts as the transaction that just completed.
- Otherwise, the observer method is called in the same transaction context and lifecycle contexts as the invocation of `Event.fire()`.

9.6. Observable container lifecycle events

9.6.1. Startup event

Implementations are required to *synchronously* fire an event with payload `jakarta.enterprise.event.Startup` and qualifier `jakarta.enterprise.inject.Any` during application

initialization. This event is fired after the event with qualifier `@Initialized(ApplicationScope.class)` but before processing requests.

This event can be observed by integrators and libraries to perform any kind of early initialization as well as by users as a reliable entry point for when the CDI container is ready.

Observer methods for this event are encouraged to specify `@Priority` to determine ordering with lower priority numbers being recommended for platform/framework/library integration and higher numbers for user applications.

Applications must never manually fire any events with payload type `jakarta.enterprise.event.Startup`.

9.6.2. Shutdown event

Implementations are required to *synchronously* fire an event with payload `jakarta.enterprise.event.Shutdown` and qualifier `jakarta.enterprise.inject.Any` during application shutdown. This event is fired during CDI container shutdown but not later than the event with qualifier `@BeforeDestroyed(ApplicationScoped.class)`.

This event can be observed by integrators and libraries to perform any kind of pre-shutdown operation as well as by users as a reliable entry point for when the CDI container is about to shut down.

Observer methods for this event are encouraged to specify `@Priority` to determine ordering with lower priority numbers being recommended for user applications and higher numbers for platform/framework/library integration.

Applications must never manually fire any events with payload type `jakarta.enterprise.event.Shutdown`.

Chapter 10. Method invokers

CDI-based frameworks often need to invoke application methods declared on managed beans. Frameworks cannot invoke application methods directly, because they are not compiled against the application code. However, during application deployment, frameworks may observe application methods through CDI extensions and build an `Invoker` for each relevant method. The invokers can then be used at application runtime to invoke the methods indirectly.

Method invokers are not supposed to be used by application code, as applications may invoke their own methods directly.

10.1. Building an `Invoker`

The CDI container allows building an `Invoker` for a method of an enabled managed bean. The method for which the invoker is built is called the *target method* of the invoker, and the managed bean is called the *target bean* of the invoker.

Invalid target methods are:

- private methods,
- constructors,
- methods declared on the `java.lang.Object` class, except for the `toString()` method,
- methods that are not declared on the bean class of the target bean or inherited from its supertypes.

Attempting to build an invoker for an invalid target method leads to a deployment problem.

Attempting to build an invoker for a non-static target method declared on a type that is not present in the set of bean types of the target bean leads to non-portable behavior. When the target bean is normal scoped, attempting to build an invoker for a non-static target method declared on an unproxyable bean type of the target bean leads to non-portable behavior.

When the target bean is not a managed bean, attempting to build an invoker leads to a deployment problem. When the target bean is an interceptor, attempting to build an invoker leads to a deployment problem.

Multiple managed beans may inherit a method from a common supertype. In that case, an invoker must be built for each target bean individually. An invoker built for one target bean may not be used to invoke the target method on an instance of another target bean.

The only way to build an invoker is using the `InvokerBuilder`. An `InvokerBuilder` can only be obtained in CDI portable extensions and build compatible extensions. See Using `InvokerBuilder` for more information.

10.2. Using an `Invoker`

The `Invoker` interface contains a single method:

```
public interface Invoker<T, R> {
    R invoke(T instance, Object[] arguments) throws Exception;
}
```

Calling `invoke()` invokes the target method on given `instance` of the target bean, passing given `arguments`, and propagates back the return value or thrown exception. The `instance` and `arguments` may be contextual or non-contextual objects.

A single invoker instance may be used to perform multiple invocations of the target method, possibly on different instances of the target bean, possibly with different arguments. Invoker implementations must be thread-safe. Whether concurrent invocations of the target method are safe depends on the implementation of the target bean and is not generally guaranteed.

Whenever a direct invocation of a method on an object is a business method invocation, an indirect invocation of that method on that object through an invoker is also a business method invocation.

10.2.1. Behavior of `invoke()`

If the target method is `static`, the `instance` is ignored; by convention, it should be `null`. If the target method is not `static` and `instance` is `null`, a `RuntimeException` is thrown. If the target method is not `static` and the `instance` is not permissible for the target method, a `RuntimeException` is thrown.

The `instance` is permissible for the target method when:

- the `instance` is a contextual reference for the target bean and the bean type that declares the target method, or
- the `instance` is a contextual reference for the target bean (regardless of the bean type) and the target method is declared on an interface that is present in the set of bean types of the target bean (see Typecasting between bean types), or
- the `instance` is a non-contextual object and the class of the `instance` declares the target method or inherits it from a supertype, or
- in other, non-portable (implementation defined) cases.

Correspondence between given `arguments` and declared parameters of the target method is positional: the Nth element of the `arguments` array is passed as the Nth argument to the target method. If the target method is a variable arity method, the last element of the `arguments` array corresponds to the variable arity parameter (and therefore must be an array). When passing an argument to the method, the applicable method invocation conversion is performed.

If the target method declares no parameter, `arguments` are ignored. If the target method declares any parameter and `arguments` is `null`, `RuntimeException` is thrown. If the `arguments` array has fewer elements than the number of parameters of the target method, `RuntimeException` is thrown. If the `arguments` array has more elements than the number of parameters of the target method, the excess elements are ignored. If a method invocation conversion does not exist from the class of some of the `arguments` (or from the null type if the argument is `null`) to the declared type of the corresponding parameter of the target method, `RuntimeException` is thrown.

NOTE	The type checking and conversion rules are aligned with pre-existing mechanisms for indirect method invocations, the Java reflection API and the method handles API.

When the declared type of a parameter of the target method is not a reifiable type, callers of `Invoker.invoke()` must ensure that the corresponding argument is constructed appropriately. Otherwise, runtime failures are likely to occur.

If the target method returns normally, its return value is returned (after boxing conversion if the target method's return type is a primitive type), unless the target method is declared `void`, in which case `null` is returned. If the target method throws an exception, it is rethrown directly.

10.2.2. Example

To illustrate how method invokers work, let's take a look at an example. Say that the following bean exists in an application and has a method that you, the framework author, want to invoke indirectly:

```
@Dependent
public class MyService {
    public String hello(String name) {
        return "Hello " + name + "!";
    }
}
```

In a CDI extension, you obtain an `InvokerBuilder` for the `hello()` method and use it to build an invoker. In a portable extension (see Using `InvokerBuilder` in CDI Full), this results in an invoker which should be stored for later usage:

```
InvokerBuilder<Invoker<MyService, String>> builder = ...;
Invoker<MyService, String> invoker = builder.build();
```

In a build compatible extension (see Using `InvokerBuilder`), this results in an opaque token that materializes as an `Invoker` at application runtime:

```
InvokerBuilder<InvokerInfo> builder = ...;
InvokerInfo invoker = builder.build();
```

To call the `hello()` method through this invoker, assuming that `myService` is a contextual reference for the bean, call:

```
invoker.invoke(myService, new Object[] {"world"})
```

The return value is `"Hello world!"`.

Internally, the container will create an implementation of the invoker, equivalent to the following class:

```
public class TheInvoker implements Invoker<MyService, String> {
    public String invoke(MyService instance, Object[] arguments) {
        return instance.hello((String) arguments[0]);
    }
}
```

10.3. Using `InvokerBuilder`

`InvokerBuilder` can be obtained in build compatible extensions from `InvokerFactory.createInvoker()`:

```
public interface InvokerFactory {
    InvokerBuilder<InvokerInfo> createInvoker(BeanInfo bean, MethodInfo method);
}
```

An `InvokerFactory` may be declared as a parameter of `@Registration` extension methods.

The target bean of the created invoker is the bean represented by the `BeanInfo` object passed to `createInvoker()`. The target method of the created invoker is the method represented by the `MethodInfo` object passed to `createInvoker()`.

```
public interface InvokerBuilder<T> {
    ...

    T build();
}
```

Calling `InvokerBuilder.build()` produces an opaque token (`InvokerInfo`) that can be passed as a parameter to a `SyntheticBeanBuilder` or `SyntheticObserverBuilder` and materializes as an `Invoker` at application runtime.

10.3.1. Configuring invoker lookups

The `InvokerBuilder` allows configuring that the `instance` or any of the `arguments` passed to `Invoker.invoke()` should be ignored and a value should be looked up from the CDI container instead.

```
public interface InvokerBuilder<T> {
    InvokerBuilder<T> withInstanceLookup();
    InvokerBuilder<T> withArgumentLookup(int position);
}
```

When `withInstanceLookup()` is called on an invoker builder and the target method is not `static`, the `invoke()` method of the built invoker shall ignore the `instance` argument and instead obtain and use a contextual reference for the target bean and the bean type that declares the target method. Calling `withInstanceLookup()` on an invoker builder for a `static` target method has no effect.

When `withArgumentLookup()` is called on an invoker builder, the `invoke()` method of the built invoker shall ignore the given element of the `arguments` array and instead:

1. identify a bean according to the rules of typesafe resolution, as defined in Performing typesafe resolution, where the required type is the declared type of the corresponding parameter of the target method and the required qualifiers are all qualifiers present on the parameter, resolving ambiguities according to Unsatisfied and ambiguous dependencies;
2. obtain and use a contextual reference for the identified bean and the declared type of the parameter.

Calling `withArgumentLookup()` with `position` less than 0 or greater than or equal to the number of parameters of the target method leads to an `IllegalArgumentException`.

Configuring a lookup using `withInstanceLookup()` or `withArgumentLookup()` does not relax the requirements defined in Behavior of `invoke()`. Notably, the `arguments` array must still have an element for each argument, regardless of whether a lookup was configured for it. This means that for a target method with N parameters, the `arguments` array must always have at least N elements.

In the following paragraphs, the beans whose instances shall be obtained by `Invoker.invoke()` as a result of calling `withInstanceLookup()` and `withArgumentLookup()` are called *looked up beans*.

During deployment validation, implementations are required to identify all looked up beans for all built invokers, as described above. It is a deployment problem if an attempt to identify a looked up bean results in an unsatisfied dependency or an ambiguous dependency that is not resolvable. Implementations are permitted to remember the identified beans and not repeat the resolution process for each invocation of `Invoker.invoke()`.

All instances of `@Dependent` looked up beans obtained during `Invoker.invoke()` are destroyed before the `invoke()` method returns or throws. The order in which the instances of `@Dependent` looked up beans are destroyed is not specified.

This specification recognizes the existence of *asynchronous* methods, where the action represented by the method does not always finish when the method returns; the *completion* of the action is asynchronous to the method call. For target methods that are considered asynchronous by the CDI container, the requirement to destroy instances of `@Dependent` looked up beans is relaxed: the instances of `@Dependent` looked up beans need not be destroyed before `Invoker.invoke()` returns. It is recommended that the instances of `@Dependent` looked up beans are destroyed after the asynchronous action completes and before the completion is propagated to the caller of `Invoker.invoke()`; if an asynchronous target method completes synchronously or throws synchronously, it is recommended that the instances of `@Dependent` looked up beans are destroyed before `invoke()` returns or rethrows the exception.

CAUTION	The rules for recognizing asynchronous methods are not specified. Applications which use invokers to call asynchronous methods are therefore not portable.

> Future versions of this specification may define an API to give greater control over the invocation of asynchronous methods.
>
> Implementations that support asynchronous methods are encouraged to document the rules they follow.

The order in which instances of looked up beans are obtained during `Invoker.invoke()` in not specified.

If an exception is thrown when creating an instance of a looked up bean during `Invoker.invoke()`, the exception is rethrown.

NOTE Destroying an instance is not permitted to throw an exception. See The `Contextual` interface for more information.

Chapter 11. Programmatic access to container

The `BeanContainer` and `BeanManager` interfaces allow programmatic access to the CDI container.

`BeanContainer` provides features that can be implemented in more restricted environments. It is available in CDI Lite environment, and therefore also in CDI Full environment.

`BeanManager` extends `BeanContainer` and provides additional features. It is only available in CDI Full environment.

In CDI Lite environment, obtaining a `BeanManager` is possible, but only methods inherited from `BeanContainer` may be invoked. Invoking `BeanManager` methods that are not inherited from `BeanContainer` results in non-portable behavior.

11.1. The `BeanContainer` object

The interface `jakarta.enterprise.inject.spi.BeanContainer` provides operations for obtaining contextual references for beans, along with many other operations of use to applications.

The container provides a built-in bean with bean type `BeanContainer`, scope `@Dependent` and qualifier `@Default`. Thus, any bean may obtain an instance of `BeanContainer` by injecting it:

```
@Inject BeanContainer container;
```

The operations of `BeanContainer` may be called at any time during the execution of the application.

11.1.1. Obtaining a reference to the CDI container

Application objects sometimes interact directly with the container via programmatic API call. The abstract class `jakarta.enterprise.inject.spi.CDI` provides access to the `BeanContainer` as well providing lookup of bean instances.

```
public abstract class CDI<T> implements Instance<T> {
   public static CDI<Object> current() { ... }
   public static void setCDIProvider(CDIProvider provider);
   public abstract BeanContainer getBeanContainer();
   public abstract BeanManager getBeanManager();
}
```

An object may obtain a reference to the current container by calling `CDI.current()`. `CDI.getBeanContainer()`, as well as other methods on `CDI`, may be called after the application initialization is completed until the application shutdown starts. If methods on `CDI` are called at any other time, non-portable behavior results.

`CDI` implements `jakarta.enterprise.inject.Instance` and therefore might be used to perform

programmatic lookup as defined in The Instance interface. If no qualifier is passed to CDI.select() method, the @Default qualifier is assumed.

When CDI.current() is called, getCDI() method is called on jakarta.enterprise.inject.spi.CDIProvider.

The CDIProvider to use may be set by the application or container using the setCDIProvider() method. If the setCDIProvider() has not been called, the service provider with highest priority of the service jakarta.enterprise.inject.spi.CDIProvider declared in META-INF/services is used. The order of more than one CDIProvider with the same priority is undefined. If no provider is available an IllegalStateException is thrown.

```
public interface CDIProvider extends Prioritized {
   CDI<Object> getCDI();
   default int getPriority();
}
```

- getPriority() method is inherited from Prioritized interface and returns the priority for the CDIProvider. If this method is not implemented the default priority 0 is assumed.

11.1.2. Obtaining a contextual reference for a bean

The method BeanContainer.getReference() returns a contextual reference for a given bean and bean type, as defined in Contextual reference for a bean.

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

The first parameter is the Bean object representing the bean. The second parameter represents a bean type that must be implemented by any client proxy that is returned. The third parameter is an instance of CreationalContext that may be used to destroy any object with scope @Dependent that is created.

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

11.1.3. Obtaining a CreationalContext

An instance of CreationalContext for a certain instance of Contextual may be obtained by calling BeanContainer.createCreationalContext().

```
public <T> CreationalContext<T> createCreationalContext(Contextual<T> contextual);
```

An instance of CreationalContext for a non-contextual object may be obtained by passing a null value to createCreationalContext().

11.1.4. Obtaining a Bean by type

The method BeanContainer.getBeans() returns the set of beans which have the given required type and qualifiers and are available for injection in the module or library containing the class into which the BeanContainer was injected, according to the rules for candidates of typesafe resolution defined in Performing typesafe resolution.

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

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

If no qualifiers are passed to getBeans(), the default qualifier @Default is assumed.

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

If two instances of the same non repeating qualifier type are given, an IllegalArgumentException is thrown.

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

11.1.5. Obtaining a Bean by name

The method BeanContainer.getBeans() which accepts a string returns the set of beans which have the given bean name and are available for injection in the module or library containing the class into which the BeanContainer was injected, according to the rules of name resolution defined in Name resolution.

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

The parameter is a bean name.

11.1.6. Resolving an ambiguous dependency

The method BeanContainer.resolve() applies the ambiguous dependency resolution rules defined in Unsatisfied and ambiguous dependencies to a set of Bean s.

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

If the ambiguous dependency resolution rules fail (as defined in Unsatisfied and ambiguous dependencies, the container must throw an AmbiguousResolutionException.

BeanContainer.resolve() must return null if:

- null is passed to resolve(), or
- no beans are passed to resolve().

11.1.7. Firing an event

The method `BeanContainer.getEvent()` returns an instance of `Event` with specified type `java.lang.Object` and specified qualifier `@Default`.

```
Event<Object> getEvent();
```

The returned instance can be used like a standard `Event` as described in Events.

11.1.8. Observer method resolution

The method `BeanContainer.resolveObserverMethods()` resolves observer methods for an event according to the rules of observer resolution defined in Observer resolution.

```
public <T> Set<ObserverMethod<? super T>> resolveObserverMethods(T event, Annotation.
.. qualifiers);
```

The first parameter of `resolveObserverMethods()` is the event object. The remaining parameters are event qualifiers.

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

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

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

11.1.9. Interceptor resolution

The method `BeanContainer.resolveInterceptors()` returns the ordered list of interceptors for a set of interceptor bindings and a type of interception and which are enabled in the module or library containing the class into which the `BeanContainer` was injected, as defined in Interceptor resolution.

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

If two instances of the same non repeating 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.

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

An application may test an annotation to determine if it is a qualifier type, scope type, stereotype or interceptor binding type, or determine if a scope type is a normal scope.

```
public boolean isScope(Class<? extends Annotation> annotationType);
public boolean isNormalScope(Class<? extends Annotation> scopeType);

public boolean isQualifier(Class<? extends Annotation> annotationType);
public boolean isInterceptorBinding(Class<? extends Annotation> annotationType);
public boolean isStereotype(Class<? extends Annotation> annotationType);
```

11.1.11. Obtaining the active `Context` for a scope

The method `BeanContainer.getContext()` retrieves an active context object associated with the given scope, as defined in The active context object for a scope.

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

11.1.12. Obtaining `Context`s for a scope

The method `BeanContainer.getContexts()` retrieves all context objects, active and inactive, associated with the given scope, as defined in Scopes and contexts.

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

11.1.13. Obtain an `Instance`

The method `BeanContainer.createInstance()` returns an `Instance<Object>` to request bean instances programmatically as described in The `Instance` interface.

The returned `Instance` object can only access instances of beans that are available for injection in the module or library containing the class into which the `BeanContainer` was injected, according to the rules defined in Typesafe resolution.

```
Instance<Object> createInstance();
```

Instances of dependent scoped beans obtained with this `Instance` object must be explicitly released by calling `Instance.destroy()` method.

If no qualifier is passed to `Instance.select()` method, the `@Default` qualifier is assumed.

11.1.14. Assignability of beans and events

The methods `BeanContainer.isMatchingBean()` and `isMatchingEvent()` provide access to assignability rules defined in Typesafe resolution and Observer resolution.

```
public boolean isMatchingBean(Set<Type> beanTypes, Set<Annotation> beanQualifiers,
Type requiredType, Set<Annotation> requiredQualifiers);

public boolean isMatchingEvent(Type eventType, Set<Annotation> eventQualifiers, Type
observedEventType, Set<Annotation> observedEventQualifiers);
```

Chapter 12. Build compatible extensions

A build compatible extension may integrate with the container during deployment time, as defined in Application initialization lifecycle.

12.1. The `BuildCompatibleExtension` interface

A build compatible extension is a service provider of the `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension` interface, declared in `META-INF/services`.

```
public interface BuildCompatibleExtension {}
```

Build compatible extensions can define arbitrary `public`, non-`static`, `void`-returning methods without type parameters, annotated with one of the extension annotations. Such methods are called *extension methods*.

Extension annotations correspond to extension execution phases:

- `@Discovery`
- `@Enhancement`
- `@Registration`
- `@Synthesis`
- `@Validation`

Extension methods may declare arbitrary number of parameters. In each execution phase, different types of parameters may be declared. All the parameters will be provided by the container when the extension method is invoked. If an extension method declares a parameter of a type unsupported in the execution phase, the container treats it as a deployment problem.

For each build compatible extension, the container creates a single instance. All extension methods are invoked on the same instance.

The invocation order for extension methods may be controlled using the `@Priority` annotation. If an extension method does not have the `@Priority` annotation, the default priority `jakarta.interceptor.Interceptor.Priority.APPLICATION + 500` is assumed. If two extension methods have equal priority, the ordering between them is undefined. Note that priority only affects order of extension methods in a single phase.

If an extension method throws an exception, the container treats it as a deployment problem.

At deployment time, CDI container does not have to be running, so calling `CDI.current()` from an extension method, or attempting to access a running CDI container in any other way, results in non-portable behavior.

12.2. The `@Discovery` phase

In this phase, build compatible extensions may register additional classes to be scanned during type discovery, and register custom CDI meta-annotations.

Extension methods annotated `@Discovery` may declare parameters of the following types:

- `ScannedClasses`
- `MetaAnnotations`
- `Messages` (see The `@Validation` phase)

```
public interface ScannedClasses {
    void add(String className);
}
```

```
public interface MetaAnnotations {
    ClassConfig addQualifier(Class<? extends Annotation> annotation);
    ClassConfig addInterceptorBinding(Class<? extends Annotation> annotation);
    ClassConfig addStereotype(Class<? extends Annotation> annotation);

    void addContext(Class<? extends Annotation> scopeAnnotation, Class<? extends
AlterableContext> contextClass);
    void addContext(Class<? extends Annotation> scopeAnnotation, boolean isNormal,
Class<? extends AlterableContext> contextClass);
}
```

If the `addQualifier`, `addInterceptorBinding` or `addStereotype` method is called, the return value allows configuring meta-annotations on the qualifier, interceptor binding or stereotype annotation and its members.

12.3. The `@Enhancement` phase

In this phase, build compatible extensions may alter annotations on discovered types.

Extension methods annotated `@Enhancement` must declare exactly one parameter of one of the following types:

- `ClassConfig` or `ClassInfo`
- `MethodConfig` or `MethodInfo`
- `FieldConfig` or `FieldInfo`

```
public interface ClassConfig extends DeclarationConfig {
    ClassInfo info();

    ClassConfig addAnnotation(Class<? extends Annotation> annotationType);
```

```
    ClassConfig addAnnotation(AnnotationInfo annotation);
    ClassConfig addAnnotation(Annotation annotation);
    ClassConfig removeAnnotation(Predicate<AnnotationInfo> predicate);
    ClassConfig removeAllAnnotations();

    Collection<MethodConfig> constructors();
    Collection<MethodConfig> methods();
    Collection<FieldConfig> fields();
}

public interface ClassInfo extends DeclarationInfo {
    String name();
    String simpleName();
    PackageInfo packageInfo();
    List<TypeVariable> typeParameters();

    Type superClass();
    ClassInfo superClassDeclaration();
    List<Type> superInterfaces();
    List<ClassInfo> superInterfacesDeclarations();

    boolean isPlainClass();
    boolean isInterface();
    boolean isEnum();
    boolean isAnnotation();
    boolean isRecord();
    boolean isAbstract();
    boolean isFinal();
    int modifiers();

    Collection<MethodInfo> constructors();
    Collection<MethodInfo> methods();
    Collection<FieldInfo> fields();
    Collection<RecordComponentInfo> recordComponents();
}
```

When an extension method declares a parameter of type `ClassConfig` or `ClassInfo`, it will be called for each discovered class matching the criteria defined on the `@Enhancement` annotation. It is possible to navigate to constructors, methods and fields from a `ClassConfig` and configure them.

```
public interface MethodConfig extends DeclarationConfig {
    MethodInfo info();

    MethodConfig addAnnotation(Class<? extends Annotation> annotationType);
    MethodConfig addAnnotation(AnnotationInfo annotation);
    MethodConfig addAnnotation(Annotation annotation);
    MethodConfig removeAnnotation(Predicate<AnnotationInfo> predicate);
    MethodConfig removeAllAnnotations();

    List<ParameterConfig> parameters();
```

```
}

public interface ParameterConfig extends DeclarationConfig {
    ParameterInfo info();

    ParameterConfig addAnnotation(Class<? extends Annotation> annotationType);
    ParameterConfig addAnnotation(AnnotationInfo annotation);
    ParameterConfig addAnnotation(Annotation annotation);
    ParameterConfig removeAnnotation(Predicate<AnnotationInfo> predicate);
    ParameterConfig removeAllAnnotations();
}

public interface MethodInfo extends DeclarationInfo {
    String name();
    List<ParameterInfo> parameters();
    Type returnType();
    Type receiverType();
    List<Type> throwsTypes();
    List<TypeVariable> typeParameters();

    boolean isConstructor();
    boolean isStatic();
    boolean isAbstract();
    boolean isFinal();
    int modifiers();

    ClassInfo declaringClass();
}

public interface ParameterInfo extends DeclarationInfo {
    String name();
    Type type();

    MethodInfo declaringMethod();
}
```

When an extension method declares a parameter of type `MethodConfig` or `MethodInfo`, it will be called for each method and constructor of each discovered class matching the criteria defined on the `@Enhancement` annotation. It is possible to navigate to method parameters from a `MethodConfig` and configure them.

```
public interface FieldConfig extends DeclarationConfig {
    FieldInfo info();

    FieldConfig addAnnotation(Class<? extends Annotation> annotationType);
    FieldConfig addAnnotation(AnnotationInfo annotation);
    FieldConfig addAnnotation(Annotation annotation);
    FieldConfig removeAnnotation(Predicate<AnnotationInfo> predicate);
    FieldConfig removeAllAnnotations();
}
```

```
public interface FieldInfo extends DeclarationInfo {
    String name();
    Type type();

    boolean isStatic();
    boolean isFinal();
    int modifiers();

    ClassInfo declaringClass();
}
```

When an extension method declares a parameter of type `FieldConfig` or `FieldInfo`, it will be called for each field of each discovered class matching the criteria defined on the `@Enhancement` annotation.

Additionally, extension methods annotated `@Enhancement` may declare parameters of the following types:

- `Types`
- `Messages` (see The `@Validation` phase)

```
public interface Types {
    Type of(Class<?> clazz);
    VoidType ofVoid();
    PrimitiveType ofPrimitive(PrimitiveType.PrimitiveKind kind);
    ClassType ofClass(String name);
    ClassType ofClass(ClassInfo clazz);
    ArrayType ofArray(Type elementType, int dimensions);
    ParameterizedType parameterized(Class<?> genericType, Class<?>... typeArguments);
    ParameterizedType parameterized(Class<?> genericType, Type... typeArguments);
    ParameterizedType parameterized(ClassType genericType, Type... typeArguments);
    WildcardType wildcardWithUpperBound(Type upperBound);
    WildcardType wildcardWithLowerBound(Type lowerBound);
    WildcardType wildcardUnbounded();
}
```

The `Types` interface allows creating representations of the void pseudo-type, primitive types, class types, array types, parameterized types and wildcard types.

To create instances of `AnnotationInfo`, `AnnotationBuilder` can be used.

12.4. The `@Registration` phase

In this phase, build compatible extensions may observe registered beans and observers.

Extension methods annotated `@Registration` must declare exactly one parameter of one of the following types:

- `BeanInfo`
- `InterceptorInfo`
- `ObserverInfo`

```
public interface BeanInfo {
    ScopeInfo scope();
    Collection<Type> types();
    Collection<AnnotationInfo> qualifiers();
    ClassInfo declaringClass();
    boolean isClassBean();
    boolean isProducerMethod();
    boolean isProducerField();
    boolean isSynthetic();
    MethodInfo producerMethod();
    FieldInfo producerField();
    boolean isAlternative();
    Integer priority();
    String name();
    DisposerInfo disposer();
    Collection<StereotypeInfo> stereotypes();
    Collection<InjectionPointInfo> injectionPoints();
}
```

When an extension method declares a parameter of type `BeanInfo`, it will be called for each bean whose set of bean types matches the criteria defined on the `@Registration` annotation.

```
public interface InterceptorInfo extends BeanInfo {
    Collection<AnnotationInfo> interceptorBindings();
    boolean intercepts(InterceptionType interceptionType);
}
```

When an extension method declares a parameter of type `InterceptorInfo`, it will be called for each interceptor whose set of bean types matches the criteria defined on the `@Registration` annotation.

```
public interface ObserverInfo {
    Type eventType();
    Collection<AnnotationInfo> qualifiers();
    ClassInfo declaringClass();
    MethodInfo observerMethod();
    ParameterInfo eventParameter();
    BeanInfo bean();
    boolean isSynthetic();
    int priority();
    boolean isAsync();
    Reception reception();
    TransactionPhase transactionPhase();
```

```
}
```

When an extension method declares a parameter of type `ObserverInfo`, it will be called for each observer whose observed event type matches the criteria defined on the `@Registration` annotation.

Additionally, extension methods annotated `@Registration` may declare parameters of the following types:

- `InvokerFactory`
- `Types`
- `Messages` (see The `@Validation` phase)

```
public interface InvokerFactory {
    InvokerBuilder<InvokerInfo> createInvoker(BeanInfo bean, MethodInfo method);
}
```

The `InvokerFactory` interface allows creating an `InvokerBuilder` for given target bean and target method (see Using `InvokerBuilder`).

12.5. The `@Synthesis` phase

In this phase, build compatible extensions may register synthetic beans and observers.

Extension methods annotated `@Synthesis` may declare parameters of the following types:

- `SyntheticComponents`
- `Types`
- `Messages` (see The `@Validation` phase)

```
public interface SyntheticComponents {
    <T> SyntheticBeanBuilder<T> addBean(Class<T> beanClass);
    <T> SyntheticObserverBuilder<T> addObserver(Class<T> eventType);
    <T> SyntheticObserverBuilder<T> addObserver(Type eventType);
}
```

The `SyntheticBeanBuilder` and `SyntheticObserverBuilder` interfaces are used to configure:

- bean or observer attributes, such as scope, bean types, qualifiers, or observed event type;
- class of a bean creation/destruction function or observer notification function;
- a string-keyed parameter map.

The container creates an instance of the bean creation/destruction function or observer notification function whenever it needs to create an instance of the bean, destroy the instance of the bean, or notify the observer. When invoking the bean creation/destruction function or observer notification function, the container passes the parameter map to it.

The parameter map may contain values of the following types:

- `boolean`
- `int`
- `long`
- `double`
- `String`
- `Class`
- `Enum`
- `Invoker`
- any annotation type
- array of any previously mentioned type

When defining the parameter map on `SyntheticBeanBuilder` or `SyntheticObserverBuilder`, it is possible to use `ClassInfo`, `InvokerInfo`, or `AnnotationInfo` to define parameter values. When such parameter is looked up from the parameter map in the synthetic bean creation/destruction function or the synthetic observer notification function, the value will be of type `Class`, `Invoker`, or the respective annotation type.

12.6. The `@Validation` phase

In this phase, build compatible extensions may perform custom validation.

Extension methods annotated `@Validation` may declare parameters of the following types:

- `Types`
- `Messages`

```
public interface Messages {
    void info(String message);
    void info(String message, AnnotationTarget relatedTo);
    void info(String message, BeanInfo relatedTo);
    void info(String message, ObserverInfo relatedTo);

    void warn(String message);
    void warn(String message, AnnotationTarget relatedTo);
    void warn(String message, BeanInfo relatedTo);
    void warn(String message, ObserverInfo relatedTo);

    void error(String message);
    void error(String message, AnnotationTarget relatedTo);
    void error(String message, BeanInfo relatedTo);
    void error(String message, ObserverInfo relatedTo);
    void error(Exception exception);
```

```
}
```

Calling any of the `Messages.error()` methods registers a deployment problem.

Chapter 13. Packaging and deployment

At deployment time, the container must perform *bean discovery*, execute build compatible extensions, and detect definition errors and deployment problems. The term *deployment time* in CDI Lite means before the application is started, such as during application compilation, or during application startup at latest.

Bean discovery is the process of determining:

- Bean archives within application, and any beans contained within them
- Which alternatives and interceptors are *enabled*
- The *ordering* of enabled interceptors

Additional beans may be registered programmatically using build compatible extensions.

13.1. Bean archives

Bean classes of enabled beans must be deployed in *bean archives*.

A bean archive has a *bean discovery mode* of either `annotated` or `none`. This is governed by presence of `beans.xml` file which can be either empty or it can declare the `bean-discovery-mode` attribute. Default value for this attribute is `annotated`.

An archive which:

- contains a `beans.xml` file with the `bean-discovery-mode` of `none`, or,
- contains a portable extension or a build compatible extension and no `beans.xml` file

is not a bean archive.

An *implicit bean archive* is:

- an archive which contains a `beans.xml` file that is empty, or,
- an archive which contains a `beans.xml` file that has `bean-discovery-mode` attribute set to `annotated`

Any other archive which contains a `beans.xml` file is not portable in CDI Lite. More kinds of bean archives exist in CDI Full.

Implementations that do not support CDI Full are required to ignore the content of the `beans.xml` file, except for the `bean-discovery-mode` attribute. Implementations that do not support CDI Full are required to detect presence of an archive which contains a `beans.xml` file that has `bean-discovery-mode` attribute set to `all` and treat it as a deployment problem.

To ensure portability between CDI Lite and CDI Full, applications are encouraged to:

- always add a `beans.xml` file to an archive which contains classes with bean defining annotations;
- never add classes with bean defining annotations to an archive without `beans.xml`.

When determining which archives are bean archives, the container must consider all archives that constitute the application. Implementations are encouraged to document how the candidate archives are found in more detail.

The `beans.xml` file must be named:

- `META-INF/beans.xml`.

The container searches for beans in all bean archives discovered as described above.

If a bean class is deployed in two different bean archives, non-portable behavior results. Portable applications must deploy each bean class in no more than one bean archive.

Implicit bean archives are likely to contain classes which are not deployed as beans.

An extension may be deployed in any archive, including those that are not bean archives.

13.2. Deployment

At deployment time, the container performs the following steps:

- First, the container must perform type discovery, as defined in Type discovery. As part of that, the container must execute the `@Discovery` and `@Enhancement` phases of build compatible extensions.
- Next, the container must perform bean discovery, as defined in Bean discovery. As part of that, the container must execute the `@Registration` and `@Synthesis` phases of build compatible extensions.
- Finally, the container must detect deployment problems by validating bean dependencies and invoker lookups. As part of that, the container must execute the `@Validation` phase of build compatible extensions.

At any step, the container must abort deployment if any definition errors or deployment problems exist, as defined in Problems detected automatically by the container.

13.3. Application initialization lifecycle

CDI Lite does not require the container to perform any other initialization during application startup. With deployment complete, the container begins directing requests to the application.

13.4. Application shutdown lifecycle

When an application is stopped, the container must destroy all contexts.

13.5. Type and Bean discovery

The container automatically discovers managed beans (according to the rules of Which Java classes are managed beans?) in bean archives and searches the bean classes for producer methods, producer fields, disposer methods and observer methods.

13.5.1. Type discovery

First the container must discover types. The container discovers each Java class with a bean defining annotation in an implicit bean archive.

The container must also execute the `@Discovery` phase of build compatible extensions and discover all classes added using the `ScannedClasses` API.

When all types are discovered, the container must execute the `@Enhancement` phase of build compatible extensions and alter its metadata representation of discovered types accordingly.

13.5.2. Bean discovery

For every type in the set of discovered types (as defined in Type discovery), the container must:

- inspect the type metadata to determine if it is a bean, and then
- detect definition errors by validating the class and its metadata, and then
- determine which alternatives and interceptors are enabled, according to the rules defined in Enabled and disabled beans.

For each enabled bean, the container must search the class for producer methods and fields, as defined in Producer methods and in Producer fields, including resources, and for disposer methods as defined in Disposer methods, and for observer methods.

Then, the container registers the `Bean` and `ObserverMethod` objects:

- For each enabled bean that is not an interceptor, the container registers an instance of the `Bean` interface defined in The `Bean` interface.
- For each enabled interceptor, the container registers an instance of the `Interceptor` interface defined in The `Interceptor` interface.
- For each observer method of every enabled bean, the container registers an instance of the `ObserverMethod` interface defined in The `ObserverMethod` interface.

The container must execute the `@Registration` phase of build compatible extensions for each registered bean, interceptor, and observer method.

Next, the container must execute the `@Synthesis` phase of build compatible extensions. For each registered synthetic bean, the container registers an instance of the `Bean` interface. For each registered synthetic observer, the container registers an instance of the `ObserverMethod` interface.

Finally, the container must execute the `@Registration` phase of build compatible extensions for each synthetic bean and synthetic observer method.

Part I.B - CDI Full

CDI Full contains all the functionality defined in CDI Lite and adds some additional features such as specialization, decorators, session scope or conversation scope. Some of these concepts were briefly mentioned in the previous CDI Lite chapter and this section of specification defines them in depth.

All rules from the CDI Lite specification apply to CDI Full, unless the CDI Full specification says otherwise. Most sections of the CDI Full specification add new rules on top of the CDI Lite specification, but some override the corresponding section of the CDI Lite specification and provide a replacing set of rules.

Chapter 14. Scopes in CDI Full

14.1. Built-in scope types in CDI Full

In addition to built-in scope types defined in Built-in scope types, the following two built-in scopes are present:

- The `@SessionScoped` annotation represents the session scope defined in Session context lifecycle.
- The `@ConversationScoped` annotation represents the conversation scope defined in Conversation context lifecycle.

In addition to rules defined in Built-in scope types, the following rules apply.

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

14.2. Bean defining annotations in CDI Full

In addition to bean defining annotations defined in Bean defining annotations, the following bean defining annotations are present:

- `@SessionScoped` and `@ConversationScoped` annotations,
- `@Decorator` annotation.

14.2.1. Built-in stereotypes in CDI Full

In addition to built-in stereotypes defined in Built-in stereotypes, the following built-in stereotype is present.

The special-purpose `@Decorator` stereotype is defined in Declaring a decorator.

Chapter 15. Inheritance and specialization in CDI Full

15.1. Specializing a managed bean

In addition to rules defined in Managed beans, the following rules apply.

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 Specialization.

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

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

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

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

15.2. Specializing a producer method

In addition to rules defined in Producer methods, the following rules apply.

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 Specialization.

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

```
@Mock
public class MockShop extends Shop {

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

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

```
    ...

}
```

15.3. Specialization

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

Consider the following beans:

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

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

Suppose that the `MockAsynchronousService` alternative is selected, as defined in Modularity:

```
@Alternative @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION+100)
public class MockAsynchronousService extends AsynchronousService {
    ...
}
```

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

```
@Inject Service service;
```

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

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

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

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

To help prevent developer error, the first bean may:

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

explicitly declare that it *specializes* the second bean.

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

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

15.3.1. Direct and indirect specialization

The annotation `@jakarta.enterprise.inject.Specializes` is used to indicate that one bean *directly specializes* another bean, as defined in Specializing a managed bean and Specializing a producer method.

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

- direct specialization, where X directly specializes Y, or
- transitive specialization, where a bean Z exists, such that X directly specializes Z and Z specializes Y.

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

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

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

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

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

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

If `AsynchronousService` declared a bean name:

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

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

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

Chapter 16. Dependency injection and lookup in CDI Full

16.1. Modularity in CDI Full

In addition to rules defined in Modularity, the following rules apply.

A library may be an explicit bean archive or an implicit bean archive, as defined in Bean archives in CDI Full.

An alternative is not available for injection, lookup or name resolution to classes in a module unless the module is a bean archive and the alternative is explicitly *selected* for the bean archive or the application.

16.1.1. Declaring selected alternatives in CDI Full

CDI Full provides an additional way to select alternatives to the one defined in Declaring selected alternatives for an application.

16.1.1.1. Declaring selected alternatives for an application in CDI Full

In addition to rules defined in Declaring selected alternatives for an application, the following rule applies.

Custom bean implementations which are also alternatives may implement Prioritized interface in which case they will be enabled for entire application with given priority.

16.1.1.2. Declaring selected alternatives for a bean archive

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

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

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
        version="3.0">
   <alternatives>
      <class>com.acme.myfwk.InMemoryDatabase</class>
      <stereotype>com.acme.myfwk.Mock</stereotype>
      <stereotype>com.acme.site.Australian</stereotype>
```

```
    </alternatives>
</beans>
```

For each child `<class>` element, the container verifies that either:

- a class with the specified name exists and is annotated with `@Alternative` or an `@Alternative` stereotype, or
- a class with the specified name exists and declares a field or method annotated with `@Produces` and, at the same time, annotated with `@Alternative` or an `@Alternative` stereotype, or
- an alternative bean whose bean class has the specified name exists.

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

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

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

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

16.1.2. Enabled and disabled beans in CDI Full

The rules defined in Enabled and disabled beans are overridden as follows.

A bean is said to be *enabled* if:

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

Otherwise, the bean is said to be disabled.

16.1.3. Inconsistent specialization

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

16.1.4. Inter-module injection in CDI Full

Instead of the rules in Inter-module injection, the following rules apply in CDI Full.

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

- the bean is not an interceptor or decorator,
- the bean is enabled,
- the bean is either not an alternative, or the module is a bean archive and the bean is a selected alternative of the bean archive, or the bean is a selected alternative of the application, and
- the bean class is required to be accessible to classes in the module, according to the class accessibility requirements of the module architecture.

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

16.2. Typesafe resolution in CDI Full

16.2.1. Performing typesafe resolution in CDI Full

In addition to rules defined in Performing typesafe resolution, the following rules apply.

- 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 Assignability of raw and parameterized types or Assignability of raw and parameterized types for delegate injection points.

Furthermore, for a custom implementation of the `Bean` interface defined in The `Bean` interface, the container calls `getTypes()` and `getQualifiers()` to determine the bean types and qualifiers.

16.2.2. Unsatisfied and ambiguous dependencies in CDI Full

In addition to rules defined in Unsatisfied and ambiguous dependencies, the following rules apply.

An unsatisfied or ambiguous dependency cannot exist for a decorator delegate injection point, defined in Decorator delegate injection points.

Furthermore, for a custom implementation of the `Bean` interface defined in The `Bean` interface, the container calls `getInjectionPoints()` to determine the set of injection points.

16.2.3. Assignability of raw and parameterized types in CDI Full

In addition to rules defined in Assignability of raw and parameterized types, the following rules apply.

A special set of rules, defined in Assignability of raw and parameterized types for delegate injection points, apply if and only if the injection point is a decorator delegate injection point.

16.2.4. Ambiguous names in CDI Full

In addition to rules defined in Ambiguous names, the following rules apply.

When an ambiguous name exists, the container attempts to resolve the ambiguity. The container eliminates all eligible beans that are not alternatives selected for the bean archive or selected for the application, except for producer methods and fields of beans that are alternatives.

16.3. Client proxies in CDI Full

In addition to the reasons for indirection defined in Client proxies, in CDI Full client proxies may be passivated, even when the bean itself may not be. Therefore, the container must use a client proxy whenever a bean with normal scope is injected into a bean with a passivating scope, as defined in Passivation and passivating scopes. (On the other hand, beans with scope `@Dependent` must be serialized along with their client.)

16.4. Dependency injection in CDI Full

16.4.1. Injection point metadata in CDI Full

The behavior of `InjectionPoint` metadata is overridden as follows:

- The `getAnnotated()` method returns an instance of `jakarta.enterprise.inject.spi.AnnotatedField` or `jakarta.enterprise.inject.spi.AnnotatedParameter`, depending upon whether the injection point is an injected field or a constructor/method parameter. If the injection point represents a dynamically obtained instance, then the `getAnnotated()` method returns an instance of `jakarta.enterprise.inject.spi.AnnotatedField` or `jakarta.enterprise.inject.spi.AnnotatedParameter` representing the `Instance` injection point, depending upon whether the injection point is an injected field or a constructor/method parameter.
- The `isDelegate()` method returns `true` if the injection point is a decorator delegate injection point, and `false` otherwise. If the injection point represents a dynamically obtained instance then `isDelegate()` returns false.

If the injection point represents a dynamically obtained instance then the `isTransient()` method returns `true` if the `Instance` injection point is a transient field, and `false` otherwise. If this injection point is declared as transient, after bean's passivation, the value will not be restored. Instance<> injection point is the preferred approach.

In CDI Full, the built-in implementation of `InjectionPoint` must be a passivation capable dependency, as defined in Passivation capable dependencies.

16.4.2. Bean metadata in CDI Full

In addition to rules defined in Bean metadata, the following rules apply.

The interfaces `Decorator` also provides metadata about a bean.

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

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

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

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

These beans are passivation capable dependencies, as defined in Passivation capable dependencies.

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

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

If:

- the injection point is a field, an initializer method parameter or a bean constructor, with qualifier `@Default`, then the type parameter of the injected `Decorator` must be the same as the type declaring the injection point, or
- the injection point is a field, an initializer method parameter or a bean constructor of a decorator, with qualifier `@Decorated`, then the type parameter of the injected `Bean` must be the same as the delegate type.

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

16.5. Programmatic lookup in CDI Full

16.5.1. The `Instance` interface in CDI Full

16.5.2. The built-in `Instance` in CDI Full

In addition to rules defined in The built-in `Instance`, the built-in implementation of `Instance` must be a passivation capable dependency, as defined in Passivation capable dependencies.

Chapter 17. Scopes and contexts in CDI Full

17.1. The `Contextual` interface in CDI Full

In addition to rules defined in The `Contextual` interface, the following rule applies.

Portable extensions may define implementations of the `Contextual` interface that do not extend `Bean`.

17.2. The `Context` interface in CDI Full

In addition to rules defined in The `Context` interface, the following rules apply.

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

The `Context` interface may be called by portable extensions.

A context object may be defined for any of the built-in scopes and registered with the container using the `AfterBeanDiscovery` event as described in `AfterBeanDiscovery` event.

17.3. Dependent pseudo-scope in CDI Full

17.3.1. Dependent objects in CDI Full

In addition to rules defined in Dependent objects, the following rules apply.

- Instances of decorators are dependent objects of the bean instance they decorate.

17.4. Contextual instances and contextual references in CDI Full

17.4.1. Contextual instance of a bean in CDI Full

In addition to rules defined in Contextual instance of a bean, the following rule applies.

For a custom implementation of the `Bean` interface defined in The `Bean` interface, the container calls `getScope()` to determine the bean scope.

17.5. 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*.

17.5.1. 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.

- A managed bean is passivation capable if and only if the bean class is serializable and all interceptors and decorators of the bean are passivation capable.
- A producer method is passivation capable if and only if it never returns a value which is not passivation capable at runtime.
- A producer field is passivation capable if and only if it never refers to a value which is not passivation capable at runtime.

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

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

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

17.5.2. Passivation capable injection points

We call an injection point of a bean *passivation capable* if the injection point is:

- a transient field, or
- a non-transient field which resolves to a bean that is a passivation capable dependency, or
- a bean constructor parameter which is annotated with `@TransientReference`, or
- a bean constructor parameter which resolves to a bean that is a passivation capable dependency, or
- a method parameter which is annotated with `@TransientReference`, or
- a method parameter which resolves to a bean that is a passivation capable dependency.

17.5.3. Passivation capable dependencies

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

The container must guarantee that:

- all beans with normal scope are passivation capable dependencies,
- all passivation capable beans with scope `@Dependent` are passivation capable dependencies,

- 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`.

17.5.4. Passivating scopes

A *passivating scope* requires that:

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

Passivating scopes must be explicitly declared `@NormalScope(passivating=true)`.

For example, the built-in session and conversation scopes defined in Context management for built-in scopes in CDI Full are passivating scopes. No other built-in scopes are passivating scopes.

17.5.5. Validation of passivation capable beans and dependencies

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

If a managed bean which declares a passivating scope, or a built-in bean:

- is not passivation capable,
- has an injection point that is not passivation capable,
- has an interceptor or decorator that is not passivation capable,
- has an interceptor or decorator with an injection point that is not passivation capable,

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

If a producer method declares a passivating scope and:

- has a return type that is declared final and does not implement or extend `Serializable`, or,
- has an injection point that is not passivation capable

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

If a producer method declares a passivating scope and doesn't only return `Serializable` types at runtime, then the container must throw an `IllegalProductException`.

If a producer field declares a passivating scope and has a type that is declared final and does not implement or extend `Serializable` then the container automatically detects the problem and treats it as a deployment problem.

If a producer field declares a passivating scope and doesn't only contain `Serializable` values at runtime then the container must throw an `IllegalProductException`.

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

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

If a managed bean which declares a passivating scope type, has a decorator or interceptor which is not a passivation capable dependency, the container automatically detects the problem and treats it as a deployment problem.

17.6. Context management for built-in scopes in CDI Full

17.6.1. Session context lifecycle

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

17.6.2. Conversation context lifecycle

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

17.6.3. The `Conversation` interface

The container provides a built-in bean with bean type `Conversation`, scope `@RequestScoped`, and qualifier `@Default`, named `jakarta.enterprise.context.conversation`.

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

- `begin()` marks the current transient conversation long-running. A conversation identifier may, optionally, be specified. If no conversation identifier is specified, an identifier is generated by the container.
- `end()` marks the current long-running conversation transient.
- `getId()` returns the identifier of the current long-running conversation, or a null value if the current conversation is transient.

- `getTimeout()` returns the timeout, in milliseconds, of the current conversation.
- `setTimeout()` sets the timeout of the current conversation.
- `isTransient()` returns `true` if the conversation is marked transient, or `false` if it is marked long-running.

If any method of `Conversation` is called when the conversation scope is not active, a `ContextNotActiveException` is thrown.

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

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

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

17.7. Context management for custom scopes in CDI Full

In addition to rules defined in Context management for custom scopes, the following rule applies.

A portable extension may define a custom context object for built-in scopes and custom scopes. For example, a remoting framework might provide a request context object for the built-in request scope.

Chapter 18. Lifecycle of contextual instances in CDI Full

18.1. Container invocations and interception in CDI Full

Instead of the rules in Container invocations and interception, the following rules apply in CDI Full.

When the application invokes:

- a method of a bean via a contextual reference to the bean, as defined in Contextual reference for a bean,
- a method of a bean via a non-contextual reference to the bean, if the instance was created by the container (e.g. using `InjectionTarget.produce()` or `UnmanagedInstance.produce()`, or
- a method of a bean via a non-contextual reference to the bean, if the instance was enhanced with the `InterceptionFactory` interface as defined in The `InterceptionFactory` interface,

the invocation is treated as a *business method invocation*.

When the container invokes a method of a bean, the invocation may or may not be treated as a business method invocation:

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

A method invocation passes through method interceptors and decorators if, and only if, it is a business method invocation.

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

Chapter 19. Interceptor bindings in CDI Full

This specification defines various extensions to the Jakarta Interceptors specification, including how to override the interceptor order defined by the `@Priority` annotation.

CDI Full implementations are required to support the entire Jakarta Interceptors specification, including:

- associating interceptors with classes and methods using the `@jakarta.interceptor.Interceptors` annotation,
- declaring `@AroundInvoke` interceptor methods on *target classes* (i.e. on beans).

Furthermore, CDI Full implementations are required to support additional features, including:

- custom implementations of `Interceptor`,
- usage of `InterceptionFactory` as described in The `InterceptionFactory` interface,
- enablement and ordering of interceptors per bean archive via `beans.xml` as described in Interceptor enablement and ordering in CDI Full.

19.1. Binding an interceptor to a bean in CDI Full

In addition to rules defined in Binding an interceptor to a bean, the following rules apply.

Interceptor bindings may be used to associate interceptors with any managed bean that is not a decorator.

It is possible to apply interceptors programmatically to the return value of a producer method, with the `InterceptionFactory` interface as defined in The `InterceptionFactory` interface.

19.2. Interceptor enablement and ordering in CDI Full

This specification extends the Jakarta Interceptors specification and defines:

- support for enabling interceptors only for a bean archive, as defined by Contexts and Dependency Injection 1.0, and
- the ability to override the interceptor order using the portable extension SPI, defined in `AfterTypeDiscovery` event.

An interceptor may be explicitly enabled for a bean archive by listing its class under the `<interceptors>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
       version="3.0">
   <interceptors>
```

```
      <class>com.acme.myfwk.TransactionInterceptor</class>
      <class>com.acme.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.

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

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

Interceptors enabled using `@Priority` are called before interceptors enabled using `beans.xml`.

An interceptor is said to be **enabled** if it is enabled in at least one bean archive or is listed in the final list of interceptors for the application, as defined in `AfterTypeDiscovery` event.

If an interceptor is enabled for the application and for the bean archive, then the enablement from the bean archive is ignored by the container. The interceptor will only be executed once based on the `@Priority` annotation's invocation chain.

19.3. Interceptor resolution in CDI Full

In addition to rules defined in Interceptor resolution, the following rules apply.

This specification extends the Jakarta Interceptors specification and defines how the interceptor bindings of a custom implementation of the `Interceptor` interface are determined.

For a custom implementation of the `Interceptor` interface defined in The `Interceptor` interface, the container calls `getInterceptorBindings()` to determine the interceptor bindings of the interceptor and `intercepts()` to determine if the interceptor intercepts a given kind of lifecycle callback or business method.

A custom implementation of the `Interceptor` interface may implement the `Prioritized` interface to be enabled for the entire application with a priority value.

Chapter 20. Decorators

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

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

Decorators may be associated with any managed bean that is not itself an interceptor or decorator, or with any built-in bean other than the built-in bean with type `BeanManager` and qualifier `@Default`. Decorators are not automatically applied to the return value of a producer method or the current value of a producer field.

A decorator instance is a dependent object of the object it decorates.

20.1. Decorator beans

A decorator is a managed bean. The set of decorated types of a decorator includes all bean types of the managed bean which are Java interfaces, except for `java.io.Serializable`. The decorator bean class and its superclasses are not decorated types of the decorator. The decorator class may be abstract.

If the set of decorated types of a decorator is empty, the container automatically detects the problem and treats it as a definition error.

If a decorator declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

20.1.1. Declaring a decorator

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

```
@Decorator @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION)
class TimestampLogger implements Logger { ... }
```

20.1.2. Decorator delegate injection points

All decorators have a *delegate injection point*. A delegate injection point is an injection point of the bean class. The type and qualifiers of the injection point are called the *delegate type* and *delegate qualifiers*. The decorator applies to beans that are assignable to the delegate injection point.

The delegate injection point must be declared by annotating the injection point with the annotation `@jakarta.decorator.Delegate`:

```
@Decorator @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION)
```

```
class TimestampLogger implements Logger {
    @Inject @Delegate @Any Logger logger;
    ...
}
```

```
@Decorator @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION)
class TimestampLogger implements Logger {
    private Logger logger;

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

A decorator must have exactly one delegate injection point. If a decorator has more than one delegate injection point, or does not have a delegate injection point, the container automatically detects the problem and treats it as a definition error.

The delegate injection point must be an injected field, initializer method parameter or bean constructor method parameter. If an injection point that is not an injected field, initializer method parameter or bean constructor method parameter is annotated `@Delegate`, the container automatically detects the problem and treats it as a definition error.

If a bean class that is not a decorator has an injection point annotated `@Delegate`, the container automatically detects the problem and treats it as a definition error.

The container must inject a *delegate* object to the delegate injection point. The delegate object implements the delegate type and delegates method invocations to remaining uninvoked decorators and eventually to the bean. When the container calls a decorator during business method interception, the decorator may invoke any method of the delegate object.

```
@Decorator @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION)
class TimestampLogger implements Logger {
    @Inject @Delegate @Any Logger logger;

    void log(String message) {
        logger.log( timestamp() + ": " + message );
    }
    ...
}
```

If a decorator invokes the delegate object at any other time, the invoked method throws an `IllegalStateException`.

20.1.3. Decorated types of a decorator

The delegate type of a decorator must implement or extend every decorated type (with exactly the same type parameters). If the delegate type does not implement or extend a decorated type of the decorator (or specifies different type parameters), the container automatically detects the problem and treats it as a definition error.

A decorator is not required to implement the delegate type.

A decorator may be an abstract Java class, and is not required to implement every method of every decorated type. Whenever the decorator does not implement a method of the decorated type, the container will provide an implicit implementation that calls the method on the delegate. If a decorator has abstract methods that are not declared by a decorated type, the container automatically detects the problem and treats it as a definition error.

The decorator intercepts every method which is declared by a decorated type of the decorator and is implemented by the bean class of the decorator.

20.2. Decorator enablement and ordering

This specification defines two methods of enabling and ordering decorators. From Contexts and Dependency Injection 1.1 onwards the `@Priority` annotation allows a decorator to be enabled and ordered for an entire application. Contexts and Dependency Injection 1.0 allowed only for a decorator to be enabled and ordered for a bean archive.

Decorators are called after interceptors. Decorators enabled using `@Priority` are called before decorators enabled using `beans.xml`.

A decorator is said to be **enabled** if it is enabled in at least one bean archive or is listed in the final list of decorators for the application, as defined in `AfterTypeDiscovery` event.

If a decorator is enabled for the application and for the bean archive, then the enablement from the bean archive is ignored by the container. The decorator will only be executed once based on the `@Priority` annotation's invocation chain.

20.2.1. Decorator enablement and ordering for an application

A decorator may be enabled for the entire application by applying the `@Priority` annotation, along with a priority value, on the decorator class. Decorators with the smaller priority values are called first. The order of more than one decorator with the same priority is undefined.

```
@Decorator @Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION)
class TimestampLogger implements Logger {

   ...
}
```

The priority value ranges defined in the Java Interceptors specification section 5.5 should be used

when defining decorator priorities.

20.2.2. Decorator enablement and ordering for a bean archive

A decorator may be explicitly enabled by listing its bean class under the `<decorators>` element of the `beans.xml` file of the bean archive.

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
        version="3.0">
   <decorators>
      <class>com.acme.myfwk.TimestampLogger</class>
      <class>com.acme.myfwk.IdentityLogger</class>
   </decorators>
</beans>
```

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

Each child `<class>` element must specify the name of a decorator bean class. If there is no class with the specified name, or if the class with the specified name is not a decorator bean class, the container automatically detects the problem and treats it as a deployment problem.

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

20.3. Decorator resolution

The process of matching decorators to a certain bean is called *decorator resolution*. A decorator is bound to a bean if:

- The bean is assignable to the delegate injection point according to the rules defined in Typesafe resolution (using Assignability of raw and parameterized types for delegate injection points).
- The decorator is enabled in the bean archive containing the bean.

If a decorator matches a managed bean, the managed bean class must be a proxyable bean type, as defined in Unproxyable bean types.

For a custom implementation of the `Decorator` interface defined in The `Decorator` interface, the container calls `getDelegateType()`, `getDelegateQualifiers()` and `getDecoratedTypes()` to determine the delegate type and qualifiers and decorated types of the decorator.

A custom implementation of the `Decorator` interface may implement the `Prioritized` interface to be enabled for the entire application with a priority value.

20.3.1. Assignability of raw and parameterized types for delegate injection points

Decorator delegate injection points have a special set of rules for determining assignability of raw and parameterized types, as an exception to Assignability of raw and parameterized types.

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

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

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

20.4. Decorator invocation

Whenever a business method is invoked on an instance of a bean with decorators, the container intercepts the business method invocation and, after processing all interceptors of the method, invokes decorators of the bean.

The container searches for the first decorator of the instance that implements the method that is being invoked as a business method. If no such decorator exists, the container invokes the business method of the intercepted instance. Otherwise, the container calls the method of the decorator.

When any decorator is invoked by the container, it may in turn invoke a method of the delegate. The container intercepts the delegate invocation and searches for the first decorator of the instance such that:

- the decorator occurs after the decorator invoking the delegate, and
- the decorator implements the method that is being invoked upon the delegate.

If no such decorator exists, the container invokes the business method of the intercepted instance. Otherwise, the container calls the method of the decorator.

20.5. Additional decorator rules

This chapter congregates various rules and limitations that apply to decorators in regard to other chapters of the specification.

20.5.1. Bean names

In addition to rules defined in Beans with no name, the following rules apply.

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

20.5.2. Alternatives

In addition to rules defined in Declaring an alternative, the following rules apply.

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

20.6. Managed beans

In addition to rules defined in Managed beans, the following rules apply.

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

20.7. Producer methods

In addition to rules defined in Declaring a producer method, the following rules apply.

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

20.8. Producer fields

In addition to rules defined in Declaring a producer field, the following rules apply.

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

20.9. Disposer methods

In addition to rules defined in Declaring a disposer method, the following rules apply.

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

20.10. Unproxyable bean types

In addition to rules defined in Unproxyable bean types, the following rules apply.

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

- that has an associated decorator.

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

Chapter 21. Events in CDI Full

21.1. Firing events in CDI Full

21.1.1. The built-in `Event` in CDI Full

In addition to rules defined in The built-in `Event`, the following rule applies.

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

21.2. Observer resolution in CDI Full

In addition to rules defined in Observer resolution, the following rule applies.

For a custom implementation of the `ObserverMethod` interface defined in The `ObserverMethod` interface, the container must call `getObservedType()` and `getObservedQualifiers()` to determine the observed event type and qualifiers, and `isAsync()` to determine whether the observer is asynchronous or synchronous.

21.3. Observer methods in CDI Full

In addition to rules defined in Observer methods, the following rules apply.

An observer method is a non-abstract method of a portable extension, as defined in Container lifecycle events.

A portable extension may declare multiple observer methods.

21.3.1. Declaring an observer method in CDI Full

In addition to rules defined in Declaring an observer method, the following rules apply.

Decorators may not declare observer methods. If a decorator has a method with a parameter annotated `@Observes` or `@ObservesAsync`, the container automatically detects the problem and treats it as a definition error.

21.4. Observer notification in CDI Full

In addition to rules defined in Observer notification, the following rule applies.

For a custom implementation of the `ObserverMethod` interface defined in The `ObserverMethod` interface, the container must call `getTransactionPhase()` to determine if the observer method is transactional observer method, and `notify()` which accepts `jakarta.enterprise.inject.spi.EventContext` to invoke the method.

Chapter 22. Method invokers in CDI Full

22.1. Building an `Invoker` in CDI Full

In addition to rules defined in Building an `Invoker`, the following rules apply.

When the target bean is a decorator, attempting to build an invoker leads to a deployment problem.

22.2. Using `InvokerBuilder` in CDI Full

In addition to rules defined in Using `InvokerBuilder`, the following rules apply.

`InvokerBuilder` can be obtained in portable extensions from `ProcessManagedBean.createInvoker()`:

```
public interface ProcessManagedBean<X> extends ProcessBean<X> {
    ...

    InvokerBuilder<Invoker<X, ?>> createInvoker(AnnotatedMethod<? super X> method);
}
```

The target bean of the created invoker is the bean for which the `ProcessManagedBean` event was fired. The target method of the created invoker is the method represented by the `AnnotatedMethod` object passed to `createInvoker()`.

Calling `InvokerBuilder.build()` produces an `Invoker` which should be stored for usage at application runtime.

Chapter 23. 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

23.1. The `Bean` interface

The `BeanAttributes` interface exposes the basic attributes of a bean.

```
public interface BeanAttributes<T> {
    public Set<Type> getTypes();
    public Set<Annotation> getQualifiers();
    public Class<? extends Annotation> getScope();
    public String getName();
    public Set<Class<? extends Annotation>> getStereotypes();
    public boolean isAlternative();
}
```

- `getTypes()`, `getQualifiers()`, `getScope()`, `getName()` and `getStereotypes()` must return the bean types, qualifiers, scope type, bean name and stereotypes of the bean, as defined in Concepts.
- `isAlternative()` must return `true` if the bean is an alternative, and `false` otherwise.

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

```
public interface Bean<T> extends Contextual<T>, BeanAttributes<T> {
    public Class<?> getBeanClass();
    public Set<InjectionPoint> getInjectionPoints();
}
```

- `getBeanClass()` returns the bean class of the managed bean or of the bean that declares the producer method or field.
- `getInjectionPoints()` returns a set of `InjectionPoint` objects, defined in Injection point metadata, representing injection points of the bean, that will be validated by the container at initialization time.

Note that implementations of `Bean` must also implement the inherited operations defined by the `Contextual` interface defined in The `Contextual` interface.

An instance of `Bean` must exist for every enabled bean.

A portable extension may add support for new kinds of beans beyond those defined by the this specification by implementing `Bean` and registering beans with the container, using the mechanism defined in `AfterBeanDiscovery` event.

Custom implementations of `Bean` are encouraged to implement `PassivationCapable` and may be required to in later revisions of this specification.

23.1.1. The `Decorator` interface

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

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

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

An instance of `Decorator` exists for every enabled decorator.

23.1.2. The `Interceptor` interface

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

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

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

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

```
public enum InterceptionType {
    AROUND_INVOKE, AROUND_CONSTRUCT, POST_CONSTRUCT, PRE_DESTROY, PRE_PASSIVATE,
POST_ACTIVATE, AROUND_TIMEOUT
```

```
}
```

An instance of `Interceptor` exists for every enabled interceptor.

23.1.3. The `ObserverMethod` interface

The interface `jakarta.enterprise.inject.spi.ObserverMethod` defines everything the container needs to know about an observer method.

```
public interface ObserverMethod<T> extends Prioritized {
    public Class<?> getBeanClass();
    public Bean<?> getDeclaringBean();
    public Type getObservedType();
    public Set<Annotation> getObservedQualifiers();
    public Reception getReception();
    public TransactionPhase getTransactionPhase();
    public int getPriority();
    public void notify(T event);
    public void notify(EventContext<T> eventContext);
    public boolean isAsync();
}
```

- `getBeanClass()` returns the class of the type that declares the observer method.
- `getDeclaringBean()` returns the `Bean` object that declares the observer method. Return value is undefined for synthetic observers.
- `getObservedType()` and `getObservedQualifiers()` return the observed event type and qualifiers.
- `getReception()` returns `IF_EXISTS` for a conditional observer and `ALWAYS` otherwise.
- `getTransactionPhase()` returns the appropriate transaction phase for a transactional observer method or `IN_PROGRESS` otherwise.
- `getPriority()` this method inherited from `Prioritized` interface returns the priority that will be used by the container to determine the notification order as defined in Observer ordering. If this method is not implemented the default priority `APPLICATION + 500` is assumed.
- `notify()` calls the observer method, as defined in Observer notification.
- `isAsync()` returns `true` if the method is an asynchronous observer method otherwise returns `false`.

An instance of `ObserverMethod` must exist for every observer method of every enabled bean.

23.1.4. The `Prioritized` interface

CDI 2.0 introduced the prioritized interface to add programmatic priority to custom SPI implementation.

```
public interface Prioritized {
    int getPriority();
```

```
}
```

23.2. The `Producer` and `InjectionTarget` interfaces

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

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

For a `Producer` that represents a class:

- `produce()` calls the constructor annotated `@Inject` if it exists, or the constructor with no parameters otherwise, as defined in Injection using the bean constructor, and returns the resulting instance. If the class has interceptors, `produce()` is responsible for building the interceptors and decorators of the instance. The instance returned by `produce()` may be a proxy.
- `dispose()` does nothing.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all injected fields, bean constructor parameters and initializer method parameters.

For a `Producer` that represents a producer method or field:

- `produce()` calls the producer method on, or accesses the producer field of, a contextual instance of the bean that declares the producer method, as defined in Invocation of producer or disposer methods.
- `dispose()` calls the disposer method, if any, on a contextual instance of the bean that declares the disposer method, as defined in Invocation of producer or disposer methods, or performs any additional required cleanup, if any, to destroy state associated with a resource.
- `getInjectionPoints()` returns the set of `InjectionPoint` objects representing all parameters of the producer method.

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

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

- `inject()` performs dependency injection upon the given object. The container sets the value of

all injected fields, and calls all initializer methods, as defined in Injection of fields and initializer methods.

- `postConstruct()` calls the `@PostConstruct` callback, if it exists.
- `preDestroy()` calls the `@PreDestroy` callback, if it exists.

Implementations of `Producer` and `InjectionTarget` must ensure that the set of injection points returned by `getInjectionPoints()` are injected by `produce()` or `inject()`.

23.3. The `BeanManager` object

In addition to rules defined in The `BeanContainer` object, the following rules apply.

The interface `jakarta.enterprise.inject.spi.BeanManager` provides operations for obtaining contextual references for beans, along with many other operations of use to applications and portable extensions.

The interface `jakarta.enterprise.inject.spi.BeanManager` extends `jakarta.enterprise.inject.spi.BeanContainer`. In CDI Full environment, `BeanContainer` is subject to the same rules as `BeanManager`.

The container provides a built-in bean with bean type `BeanManager`, scope `@Dependent` and qualifier `@Default`. The built-in implementation must be a passivation capable dependency, as defined in Passivation capable dependencies. Thus, any bean may obtain an instance of `BeanManager` by injecting it:

```
@Inject BeanManager manager;
```

Note that, an exception is thrown if the following operations are called before the `AfterBeanDiscovery` event is fired:

- `getBeans(String)`,
- `getBeans(Type, Annotation…)`,
- `getPassivationCapableBean(String)`
- `resolve(Set)`,
- `resolveDecorators(Set, Annotation…)`,
- `resolveInterceptors(InterceptionType, Annotation…)`,
- `resolveObserverMethods(Object, Annotation…)`,
- `validate(InjectionPoint)`,

and if the following operations are called before the `AfterDeploymentValidation` event is fired:

- `createInstance()`,
- `getReference(Bean, Type, CreationalContext)`,
- `getInjectableReference(InjectionPoint, CreationalContext)`.

All other operations of `BeanManager` may be called at any time during the execution of the application.

23.3.1. Obtaining a reference to the CDI container in CDI Full

In addition to rules defined in Obtaining a reference to the CDI container, the following rules apply.

Portable extensions and other objects sometimes interact directly with the container via programmatic API call. The abstract `jakarta.enterprise.inject.spi.CDI` provides access to the `BeanManager` as well providing lookup of bean instances.

A portable extension or other object may obtain a reference to the current container by calling `CDI.current()`. `CDI.getBeanManager()` and `CDI.getBeanContainer()` may be called at any time after the container fires the `BeforeBeanDiscovery` container lifecycle event until the container fires the `BeforeShutdown` container lifecycle event. If methods on `CDI` are called at any other time, non-portable behavior results.

23.3.2. Obtaining an injectable reference

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

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

The first parameter represents the target injection point. The second parameter is an instance of `CreationalContext` that may be used to destroy any object with scope `@Dependent` that is created.

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

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

23.3.3. Obtaining non-contextual instance

A non-contextual instance can be obtained and injected from an `InjectionTarget`, however the `InjectionTarget` interface is designed to work on contextual instances. A helper class, `Unmanaged` provides a set of methods optimized for working with non-contextual instances.

For example:

```
Unmanaged<Foo> unmanagedFoo = new Unmanaged<Foo>(Foo.class);
```

```
UnmanagedInstance<Foo> fooInstance = unmanagedFoo.newInstance();
Foo foo = fooInstance.produce().inject().postConstruct().get();
... // Use the foo instance
fooInstance.preDestroy().dispose();
```

23.3.4. Obtaining a Bean by type in CDI Full

In addition to rules defined in Obtaining a Bean by type, the following rules apply.

The method BeanManager.getBeans() returns the set of beans which have the given required type and qualifiers and are available for injection in the module or library containing the class into which the BeanManager was injected or the class from whose JNDI environment namespace the BeanManager was obtained, according to the rules for candidates of typesafe resolution defined in Performing typesafe resolution.

23.3.5. Obtaining a Bean by name in CDI Full

In addition to rules defined in Obtaining a Bean by name, the following rules apply.

The method BeanManager.getBeans() which accepts a string returns the set of beans which have the given bean name and are available for injection in the module or library containing the class into which the BeanManager was injected or the class from whose JNDI environment namespace the BeanManager was obtained, according to the rules of name resolution defined in Name resolution.

23.3.6. Obtaining a passivation capable bean by identifier

The method BeanManager.getPassivationCapableBean() returns the PassivationCapable bean with the given identifier (see Passivation capable beans).

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

23.3.7. Validating an injection point

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

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

23.3.8. Decorator resolution

The method BeanManager.resolveDecorators() returns the ordered list of decorators for a set of bean types and a set of qualifiers and which are enabled in the module or library containing the class into which the BeanManager was injected or the class from whose JNDI environment namespace the BeanManager was obtained, as defined in Decorator resolution.

```
List<Decorator<?>> resolveDecorators(Set<Type> types, Annotation... qualifiers);
```

The first argument is the set of bean types of the decorated bean. The annotations are qualifiers declared by the decorated bean.

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

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

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

23.3.9. Interceptor resolution in CDI Full

In addition to rules defined in Interceptor resolution, the following rules apply.

The method `BeanManager.resolveInterceptors()` returns the ordered list of interceptors for a set of interceptor bindings and a type of interception and which are enabled in the module or library containing the class into which the `BeanManager` was injected or the class from whose JNDI environment namespace the `BeanManager` was obtained, as defined in Interceptor resolution.

23.3.10. Determining if an annotation is a qualifier type, scope type, stereotype or interceptor binding type in CDI Full

In addition to rules defined in Determining if an annotation is a qualifier type, scope type, stereotype or interceptor binding type, the following rules apply.

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

```
public boolean isScope(Class<? extends Annotation> annotationType);
public boolean isQualifier(Class<? extends Annotation> annotationType);
public boolean isInterceptorBinding(Class<? extends Annotation> annotationType);
public boolean isStereotype(Class<? extends Annotation> annotationType);

public boolean isNormalScope(Class<? extends Annotation> scopeType);
public boolean isPassivatingScope(Class<? extends Annotation> scopeType);
public Set<Annotation> getInterceptorBindingDefinition(Class<? extends Annotation>
qualifierType);
public Set<Annotation> getStereotypeDefinition(Class<? extends Annotation> stereotype
);
```

23.3.11. Determining the hash code and equivalence of qualifiers and interceptor bindings

A portable extension may determine if two qualifiers or two interceptor bindings are considered equivalent for the purposes of typesafe resolution, as defined in Performing typesafe resolution.

```
public boolean areQualifiersEquivalent(Annotation qualifier1, Annotation qualifier2);
public boolean areInterceptorBindingsEquivalent(Annotation interceptorBinding1,
Annotation interceptorBinding2);
```

A portable extension may determine the hash code of a qualifier or interceptor binding, ignoring any members annotated with `@Nonbinding`.

```
public int getQualifierHashCode(Annotation qualifier);
public int getInterceptorBindingHashCode(Annotation interceptorBinding);
```

23.3.12. Obtaining an `AnnotatedType` for a class

The method `BeanManager.createAnnotatedType()` returns an `AnnotatedType` that may be used to read the annotations of the given Java class or interface.

```
public <T> AnnotatedType<T> createAnnotatedType(Class<T> type);
```

23.3.13. Obtaining an `InjectionTarget` for a class

The method `BeanManager.getInjectionTargetFactory()` returns a factory capable of creating container provided implementations of `InjectionTarget` for a given `AnnotatedType` or throws an `IllegalArgumentException` if there is a definition error associated with any injection point of the type.

```
public <T> InjectionTargetFactory<T> getInjectionTargetFactory(AnnotatedType<T> type);
```

```
public interface InjectionTargetFactory<T> {

    public InjectionTarget<T> createInjectionTarget(Bean<T> bean);
    public AnnotatedTypeConfigurator<T> configure();

}
```

Null should be passed to `InjectionTargetFactory.createInjectionTarget()` to create a non-contextual injection target.

- `configure()` method returns an `AnnotatedTypeConfigurator` as defined in `AnnotatedTypeConfigurator` SPI to configure the `AnnotatedType` used to create the

`InjectionTargetFactory`. Subsequent invocations of the `configure()` method within one `InjectionTargetFactory` instance will always return the same `AnnotatedTypeConfigurator` instance. Once `createInjectionTarget()` method has been invoked, any invocations of `configure()` throws an `IllegalStateException`.

23.3.14. Obtaining a `Producer` for a field or method

The method `BeanManager.getProducerFactory()` returns a factory capable of creating container provided implementations of `Producer` for a given `AnnotatedMethod` or `AnnotatedField`, and declaring bean, or throws an `IllegalArgumentException` if there is a definition error associated with the producer method or field.

```
public <X> ProducerFactory<X> getProducerFactory(AnnotatedField<? super X> field,
Bean<X> declaringBean);
public <X> ProducerFactory<X> getProducerFactory(AnnotatedMethod<? super X> method,
Bean<X> declaringBean);
```

```
public interface ProducerFactory<X> {

    public <T> Producer<T> createProducer(Bean<T> bean);

}
```

Null should be passed to `ProducerFactory.createProducer()` to create a producer of non-contextual objects.

23.3.15. Obtaining an `InjectionPoint`

The method `BeanManager.createInjectionPoint()` returns a container provided implementation of `InjectionPoint` for a given `AnnotatedField` or `AnnotatedParameter` or throws an `IllegalArgumentException` if there is a definition error associated with the injection point.

```
public InjectionPoint createInjectionPoint(AnnotatedField<?> field);
public InjectionPoint createInjectionPoint(AnnotatedParameter<?> parameter);
```

23.3.16. Obtaining a `BeanAttributes`

The method `BeanManager.createBeanAttributes()` returns a container provided implementation of `BeanAttributes` by reading the annotations of a given `AnnotatedType` or `AnnotatedMember`, according to the rules defined in Concepts, or throws an `IllegalArgumentException` if there is a definition error associated with the declared bean attributes.

```
public <T> BeanAttributes<T> createBeanAttributes(AnnotatedType<T> type);
public BeanAttributes<?> createBeanAttributes(AnnotatedMember<?> member);
```

23.3.17. Obtaining a Bean

The method BeanManager.createBean() returns a container provided implementation of Bean. The methods accept:

- a BeanAttributes, which determines the bean types, qualifiers, scope, name and stereotypes of the returned Bean, and the return values of isAlternative(), and
- a class, which determines the return value of Bean.getClass().
- an InjectionTargetFactory, which is used to obtain an InjectionTarget. The InjectionTarget is used to create and destroy instances of the bean, to perform dependency injection and lifecycle callbacks, and which determines the return value of Bean.getInjectionPoints().

```
public <T> Bean<T> createBean(BeanAttributes<T> attributes, Class<T> beanClass,
                              InjectionTargetFactory<T> injectionTargetFactory);
```

A second version of the method is provided to create a Bean from a producer. The method accepts:

- a BeanAttributes, which determines the bean types, qualifiers, scope, name and stereotypes of the returned Bean, and the return values of isAlternative(), and
- a class, which determines the return value of Bean.getClass().
- a ProducerFactory, which is used to obtain a Producer. The Producer is used to create and destroy instances of the bean, and which determines the return value of Bean.getInjectionPoints().

```
public <T, X> Bean<T> createBean(BeanAttributes<T> attributes, Class<X> beanClass,
                              ProducerFactory<X> producer);
```

23.3.18. Obtaining the instance of an Extension

The method BeanManager.getExtension() returns the container's instance of an Extension class declared in META-INF/services, or throws an IllegalArgumentException if the container has no instance of the given class.

```
public <T extends Extension> T getExtension(Class<T> extensionClass);
```

23.3.19. Obtain an InterceptionFactory

The method BeanManager.getInterceptionFactory() returns an InterceptionFactory for the provided type as defined in The InterceptionFactory interface.

```
<T> InterceptionFactory<T> createInterceptionFactory(CreationalContext<T> ctx, Class<
T> clazz);
```

If the actual type parameter of the method is not a Java class, non-portable behavior results.

23.3.20. Obtain an `Instance` in CDI Full

In addition to rules defined in Obtain an `Instance`, the following rules apply.

The returned `Instance` object can only access instances of beans that are available for injection in the module or library containing the class into which the `BeanManager` was injected or the Jakarta EE component from whose JNDI environment namespace the `BeanManager` was obtained, according to the rules defined in Typesafe resolution.

23.4. Unified EL integration API

Since CDI version 4.1, the Unified EL integration API, which is part of the `BeanManager` API, is deprecated. The relevant methods are placed in a new interface `jakarta.enterprise.inject.spi.el.ELAwareBeanManager`, which is present in a new supplemental CDI API artifact: `jakarta.enterprise:jakarta.enterprise.cdi-el-api`.

The requirements for supporting the Unified EL integration API are defined in the Jakarta EE Platform specification.

23.5. 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 `jakarta.enterprise.inject.spi` allow a portable extension to specify metadata that overrides the annotations that exist on a bean class. The portable extension is responsible for implementing the interfaces, thereby exposing the metadata to the container.

In general, the behavior is as defined by the Java Language Specification, and only deviations from the Java Language Specification are noted.

The interface `jakarta.enterprise.inject.spi.AnnotatedType` exposes the `Class` object and members.

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

- `getConstructors()` returns all default-access, public, protected or private constructors declared for the type.
- `getMethods()` returns all default-access, public, protected or private methods declared on the type and those declared on any supertypes. The container should call `AnnotatedMethod.getJavaMember().getDeclaringClass()` to determine the type in the type hierarchy that declared the method.

- `getFields()` returns all default-access, public, protected or private fields declared on the type and those declared on any supertypes. The container should call `AnnotatedField.getJavaMember().getDeclaringClass()` to determine the type in the type hierarchy that declared the field.

When determining annotations on a type, the container must only consider the special inheritance rules defined for scope types in Inheritance of type-level metadata.

The interface `jakarta.enterprise.inject.spi.AnnotatedField` exposes the `Field` object.

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

The interface `jakarta.enterprise.inject.spi.AnnotatedMethod` exposes the `Method` object.

```
public interface AnnotatedMethod<X>
        extends AnnotatedCallable<X> {
    public Method getJavaMember();
}
```

The interface `jakarta.enterprise.inject.spi.AnnotatedConstructor` exposes the `Constructor` object.

```
public interface AnnotatedConstructor<X>
        extends AnnotatedCallable<X> {
    public Constructor<X> getJavaMember();
}
```

The interface `jakarta.enterprise.inject.spi.AnnotatedParameter` exposes the `position` of the parameter object and the declaring program element.

```
public interface AnnotatedParameter<X>
        extends Annotated {
    public int getPosition();
    public AnnotatedCallable<X> getDeclaringCallable();
}
```

The interface `jakarta.enterprise.inject.spi.AnnotatedMember` exposes the `Member` object and the `AnnotatedType` that defines the member.

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

```
    public AnnotatedType<X> getDeclaringType();
}
```

The interface `jakarta.enterprise.inject.spi.AnnotatedCallable` exposes the parameters of an invokable object.

CDI 1.1 deprecated the method `AnnotatedMember.isStatic()`. The container should instead call `AnnotatedMember.getJavaMember().getModifiers()` to determine if the member is static.

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

The interface `jakarta.enterprise.inject.spi.Annotated` exposes the overriding annotations and type declarations.

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

- `getBaseType()` returns the type of the program element.
- `getTypeClosure()` returns all types to which the base type should be considered assignable.
- `getAnnotation(Class<T>)` returns the program element annotation of the given annotation type, or a null value.
- `getAnnotations(Class<T>)` returns the program element annotations of the given annotation type, or an empty set.
- `getAnnotations()` returns all annotations of the program element.
- `isAnnotationPresent(Class<T>)` returns `true` if the program element has an annotation of the given annotation type, or `false` otherwise.

The container must use the operations of `Annotated` and its subinterfaces to discover program element types and annotations. The container must not directly call the Java Reflection API. In particular, the container must:

- call `Annotated.getBaseType()` to determine the type of an injection point, event parameter or disposed parameter,
- call `Annotated.getTypeClosure()` to determine the bean types of any kind of bean,
- call `Annotated.getAnnotations()` to determine the scope, qualifiers, stereotypes and interceptor

bindings of a bean,

- call `Annotated.isAnnotationPresent()` and `Annotated.getAnnotation()` to read any bean annotations defined by this specification, and
- call `AnnotatedType.getConstructors()`, `AnnotatedType.getMethods()` and `AnnotatedType.getFields()` to determine the members of a bean class.

23.5.1. `AnnotatedTypeConfigurator` SPI

CDI 2.0 introduced a new SPI to help defining and creating instance for type metadata.

This SPI is composed of the following interfaces:

- `jakarta.enterprise.inject.spi.configurator.AnnotatedTypeConfigurator` to configure an `AnnotatedType`
- `jakarta.enterprise.inject.spi.configurator.AnnotatedFieldConfigurator` (defined in `AnnotatedFieldConfigurator`) to configure an `AnnotatedField`
- `jakarta.enterprise.inject.spi.configurator.AnnotatedConstructorConfigurator` (defined in `AnnotatedConstructorConfigurator`) to configure an `AnnotatedConstructor`
- `jakarta.enterprise.inject.spi.configurator.AnnotatedMethodConfigurator` (defined in `AnnotatedMethodConfigurator`) to configure an `AnnotatedMethod`
- `jakarta.enterprise.inject.spi.configurator.AnnotatedParameterConfigurator` (defined in `AnnotatedParameterConfigurator`) to configure an `AnnotatedParameter`

The container must provide an implementation for each of these interfaces.

`AnnotatedTypeConfigurator` is the entry point for this SPI. Implementation of `AnnotatedTypeConfigurator` is returned by methods in the following lifecycle event:

- `BeforeBeanDiscovery` as defined in `BeforeBeanDiscovery` event
- `ProcessAnnotatedType` as defined in `ProcessAnnotatedType` event
- `AfterTypeDiscovery` as defined in `AfterTypeDiscovery` event

```
public interface AnnotatedTypeConfigurator<T> {

    AnnotatedType<T> getAnnotated();
    AnnotatedTypeConfigurator<T> add(Annotation annotation);
    AnnotatedTypeConfigurator<T> remove(Predicate<Annotation> predicate);
    AnnotatedTypeConfigurator<T> removeAll();
    Set<AnnotatedMethodConfigurator<T>> methods();
    Stream<AnnotatedMethodConfigurator<T>> filterMethods(Predicate<AnnotatedMethod<T>>
predicate);
    Set<AnnotatedFieldConfigurator<T>> fields();
    Stream<AnnotatedFieldConfigurator<T>> filterFields(Predicate<AnnotatedField<T>>
predicate);
    Set<AnnotatedConstructorConfigurator<T>> constructors();
    Stream<AnnotatedConstructorConfigurator<T>> filterConstructors(Predicate
```

```
<AnnotatedConstructor<T>> predicate);
}
```

- `getAnnotated()` returns the original `AnnotatedType` with which this configurator was initialized
- `add()` adds an annotation to the configured element
- `remove()` removes annotations that match the specified predicate from the configured element
- `removeAll()` removes all annotations from the configured element
- `methods()` returns a set of `AnnotatedMethodConfigurator` to configure annotations on methods
- `filterMethods()` returns a `Stream<AnnotatedMethodsConfigurator>` filtered by applying the provided Predicate on `methods()`
- `fields()` returns a set of `AnnotatedFieldConfigurator` to configure annotations on fields
- `filterFields()` returns a `Stream<AnnotatedFieldConfigurator>` filtered by applying the provided Predicate on `fields()`
- `constructors()` returns a set of `AnnotatedConstructorConfigurator` to configure annotations on constructors
- `filterConstructors()` returns a `Stream<AnnotatedConstructorConfigurator>` filtered by applying the provided Predicate on `Constructors()`

23.5.1.1. `AnnotatedMethodConfigurator`

`AnnotatedMethodConfigurator` is obtained through `AnnotatedTypeConfigurator` as defined in `AnnotatedTypeConfigurator` SPI

```
public interface AnnotatedMethodConfigurator<T> {
    AnnotatedMethod<T> getAnnotated();
    AnnotatedMethodConfigurator<T> add(Annotation annotation);
    AnnotatedMethodConfigurator<T> remove(Predicate<Annotation> predicate);
    AnnotatedMethodConfigurator<T> removeAll();
    List<AnnotatedParameterConfigurator<T>> params();
    Stream<AnnotatedParameterConfigurator<T>> filterParams(Predicate
<AnnotatedParameter<T>> predicate);
}
```

- `getAnnotated()` returns the original `AnnotatedMethod` with which this configurator was initialized
- `add()` adds an annotation to the configured element
- `remove()` removes annotations that match the specified predicate from the configured element
- `removeAll()` removes all annotations from the configured element
- `params()` returns a list of `AnnotatedParameterConfigurator` to configure annotations on parameters.
- `filterParams(Predicate<AnnotatedParameter<T>> predicate)` returns a `Stream<AnnotatedParameterConfigurator>` filtered by applying the provided Predicate on `params()`

23.5.1.2. AnnotatedConstructorConfigurator

AnnotatedConstructorConfigurator is obtained through AnnotatedTypeConfigurator as defined in AnnotatedTypeConfigurator SPI

```
public interface AnnotatedConstructorConfigurator<T> {
    AnnotatedConstructor<T> getAnnotated();
    AnnotatedConstructorConfigurator<T> add(Annotation annotation);
    AnnotatedConstructorConfigurator<T> remove(Predicate<Annotation> predicate);
    AnnotatedConstructorConfigurator<T> removeAll();
    List<AnnotatedParameterConfigurator<T>> params();
    Stream<AnnotatedParameterConfigurator<T>> filterParams(Predicate
<AnnotatedParameter<T>> predicate);
}
```

- getAnnotated() returns the original AnnotatedConstructor with which this configurator was initialized
- add() adds an annotation to the configured element
- remove() removes annotations that match the specified predicate from the configured element
- removeAll() removes all annotations from the configured element
- params() returns a list of AnnotatedParameterConfigurator to configure annotations on parameters.
- filterParams(Predicate<AnnotatedParameter<T>> predicate) returns a Stream<AnnotatedParameterConfigurator> filtered by applying the provided Predicate on params()

23.5.1.3. AnnotatedParameterConfigurator

AnnotatedParameterConfigurator is obtained through AnnotatedMethodConfigurator (as defined in AnnotatedMethodConfigurator) and AnnotatedConstructorConfigurator as defined in AnnotatedConstructorConfigurator.

```
public interface AnnotatedParameterConfigurator<T> {
    AnnotatedParameter<T> getAnnotated();
    AnnotatedParameterConfigurator<T> add(Annotation annotation);
    AnnotatedParameterConfigurator<T> remove(Predicate<Annotation> predicate);
    AnnotatedParameterConfigurator<T> removeAll();
}
```

- getAnnotated() returns the original AnnotatedParameter with which this configurator was initialized
- add() adds an annotation to the configured element
- remove() removes annotations that match the specified predicate from the configured element
- removeAll() removes all annotations from the configured element

23.5.1.4. `AnnotatedFieldConfigurator`

`AnnotatedFieldConfigurator` is obtained through `AnnotatedTypeConfigurator` as defined in `AnnotatedTypeConfigurator` SPI:

```
public interface AnnotatedFieldConfigurator<T> {

    AnnotatedField<T> getAnnotated();
    AnnotatedFieldConfigurator<T> add(Annotation annotation);
    AnnotatedFieldConfigurator<T> remove(Predicate<Annotation> predicate);
    AnnotatedFieldConfigurator<T> removeAll();
}
```

- `getAnnotated()` returns the original `AnnotatedField` with which this configurator was initialized
- `add()` adds an annotation to the configured element
- `remove()` removes annotations that match the specified predicate from the configured element
- `removeAll()` removes all annotations from the configured element

23.6. 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 Application initialization lifecycle in CDI Full. These events are fired synchronously.

Observer methods of these events must belong to *extensions*. An extension is a service provider of the service `jakarta.enterprise.inject.spi.Extension` declared in `META-INF/services`.

```
public interface Extension {}
```

If any method on the event object is called outside of the observer method invocation, an `IllegalStateException` is thrown.

Service providers may have observer methods, which may observe any event, including any container lifecycle event, and obtain an injected `BeanManager` reference. Any decorators associated with `BeanManager` will not be applied. If other beans are injected into an extension's observer methods, non-portable behavior results. An extension may use `BeanManager.getEvent()` to deliver events to observer methods defined on extensions. The container is not required to deliver events fired during application initialization to observer methods defined on beans.

The container instantiates a single instance of each extension at the beginning of the application initialization process and maintains a reference to it until the application shuts down. The container delivers event notifications to this instance by calling its observer methods.

If an extension declares a static observer method whose event parameter type:

- is a container lifecycle event, or

- is `java.lang.Object` and the event parameter has either no qualifiers or a single qualifier `@Any`,

non-portable behavior results.

The notification order for observer methods within extensions follows the same ordering rule as defined in Observer ordering for non-extension observers. The priority of an observer method may be declared using the `@Priority` annotation.

```
void beforeBeanDiscovery(@Observes @Priority(jakarta.interceptor.Interceptor.Priority
.LIBRARY_BEFORE) BeforeBeanDiscovery event) { ... }
```

For each service provider, the container must provide a bean of scope `@ApplicationScoped` and qualifier `@Default`, supporting injection of a reference to the service provider instance. The bean types of this bean include the class of the service provider and all superclasses and interfaces.

Lifecycle events described below can be grouped into two categories:

- Application lifecycle events, that are fired once:
  - BeforeBeanDiscovery
  - AfterTypeDiscovery
  - AfterBeanDiscovery
  - AfterDeploymentValidation
  - BeforeShutdown
- Bean discovery events, that are fired multiple times:
  - ProcessAnnotatedType
  - ProcessInjectionPoint
  - ProcessInjectionTarget
  - ProcessBeanAttributes
  - ProcessBean
  - ProcessProducer
  - ProcessObserverMethod

Note that the chronological order of these events is specified in Application initialization lifecycle in CDI Full.

As these lifecycle events are fired, the container must also execute build compatible extensions. Which phase of build compatible extensions should be executed when is indicated in the description of the corresponding lifecycle events. Build compatible extensions annotated `@SkipIfPortableExtensionPresent` must be ignored in CDI Full, if given portable extension is present.

23.6.1. `BeforeBeanDiscovery` event

The container must fire an event before it begins the type discovery process. The event object must

be of type jakarta.enterprise.inject.spi.BeforeBeanDiscovery:

```
public interface BeforeBeanDiscovery {
    public void addQualifier(Class<? extends Annotation> qualifier);
    public void addQualifier(AnnotatedType<? extends Annotation> qualifier);
    public void addScope(Class<? extends Annotation> scopeType, boolean normal,
boolean passivating);
    public void addStereotype(Class<? extends Annotation> stereotype, Annotation...
stereotypeDef);
    public void addInterceptorBinding(Class<? extends Annotation> bindingType,
Annotation... bindingTypeDef);
    public void addInterceptorBinding(AnnotatedType<? extends Annotation> bindingType
);
    public void addAnnotatedType(AnnotatedType<?> type, String id);
    public AnnotatedTypeConfigurator<?> addAnnotatedType(Class<T> type,String id);
    <T extends Annotation> AnnotatedTypeConfigurator<T> configureQualifier(Class<T>
qualifier);
    <T extends Annotation> AnnotatedTypeConfigurator<T> configureInterceptorBinding
(Class<T> bindingType);
}
```

- addQualifier() declares an annotation type as a qualifier type.
- addScope() declares an annotation type as a scope type.
- addStereotype() declares an annotation type as a stereotype, and specifies its meta-annotations.
- addInterceptorBinding() declares an annotation type as an interceptor binding type, and specifies its meta-annotations.
- addAnnotatedType() adds a given AnnotatedType to the set of types which will be scanned during bean discovery, with an optional identifier.

  Second version of the method returns a new AnnotatedTypeConfigurator as defined in AnnotatedTypeConfigurator SPI to easily configure the AnnotatedType which will be added at the end of the observer invocation. The returned AnnotatedTypeConfigurator is initialized with type and annotations of the provided class.

- configureQualifier() returns a new AnnotatedTypeConfigurator as defined in AnnotatedTypeConfigurator SPI to configure a new AnnotatedType and declares it as a qualifier type.
- configureInterceptorBinding() returns a new AnnotatedTypeConfigurator as defined in AnnotatedTypeConfigurator SPI to configure a new AnnotatedType and declares it as an interceptor binding.

```
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.

If any `BeforeBeanDiscovery` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

The container must execute the `@Discovery` phase of build compatible extensions at this time.

23.6.2. `AfterTypeDiscovery` event

The container must fire an event when it has fully completed the type discovery process and before it begins the bean discovery process. The event object must be of type `jakarta.enterprise.inject.spi.AfterTypeDiscovery`.

```
public interface AfterTypeDiscovery {
    public List<Class<?>> getAlternatives();
    public List<Class<?>> getInterceptors();
    public List<Class<?>> getDecorators();
    public void addAnnotatedType(AnnotatedType<?> type, String id);
    public AnnotatedTypeConfigurator<?> addAnnotatedType(Class<T> type,String id);
}
```

- `getAlternatives()` returns the ordered list of enabled alternatives for the application, sorted by priority in ascending order. Alternatives enabled for a bean archive are not included in the list.
- `getInterceptors()` returns the ordered list of enabled interceptors for the application, sorted by priority in ascending order. Interceptors enabled for a bean archive are not included in the list.
- `getDecorators()` returns the ordered list of enabled decorators for the application, sorted by priority in ascending order. Decorators enabled for a bean archive are not included in the list.
- `addAnnotatedType()` adds a given `AnnotatedType` to the set of types which will be scanned during bean discovery, with an identifier.

  The second version of the method, returns a new `AnnotatedTypeConfigurator` as defined in `AnnotatedTypeConfigurator` SPI to easily configure the `AnnotatedType` which will be added at the end of observer invocation. The returned `AnnotatedTypeConfigurator` is initialized with type and annotations of the provided class.

If an alternative, interceptor or decorator is added using `AfterTypeDiscovery.addAnnotatedType()`, non-portable behavior results.

Any observer of this event is permitted to add classes to, or remove classes from, the list of alternatives, list of interceptors or list of decorators. The container must use the final values of these collections, after all observers of `AfterTypeDiscovery` have been called, to determine the order of the enabled alternatives, interceptors, and decorators for application. The initial values of these collections are defined by the `@Priority` annotation.

```
void afterTypeDiscovery(@Observes AfterTypeDiscovery event) { ... }
```

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

If any `AfterTypeDiscovery` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

23.6.3. `AfterBeanDiscovery` event

The container must fire an event when it has fully completed the bean discovery process, validated that there are no definition errors relating to the discovered beans, and registered `Bean` and `ObserverMethod` objects for the discovered beans.

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

```
public interface AfterBeanDiscovery {
    public void addDefinitionError(Throwable t);
    public void addBean(Bean<?> bean);
    public BeanConfigurator<?> addBean();
    public void addObserverMethod(ObserverMethod<?> observerMethod);
    public ObserverMethodConfigurator<?> addObserverMethod();
    public void addContext(Context context);
    public <T> AnnotatedType<T> getAnnotatedType(Class<T> type, String id);
    public <T> Iterable<AnnotatedType<T>> getAnnotatedTypes(Class<T> type);
}
```

- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after all observers have been notified.
- `addBean()` fires an event of type `ProcessSyntheticBean` containing the given `Bean` and then registers the `Bean` with the container, thereby making it available for injection into other beans. The given `Bean` may implement `Interceptor` or `Decorator`.

  The second version of the method, returns a new `BeanConfigurator` as defined in `BeanConfigurator` interface to easily configure the `Bean` which will be added at the end of observer invocation. If the container is unable to process the configurator it automatically detects the problem and treats it as a deployment problem.

- `addObserverMethod()` fires an event of type `ProcessSyntheticObserverMethod` containing the given `ObserverMethod` and then registers the `ObserverMethod` with the container, thereby making it available for event notifications.

  If the given `ObserverMethod` does not override either `ObserverMethod.notify(T)` or `ObserverMethod.notify(EventContext<T>)`, the container automatically detects the problem and treats it as a definition error.

  The second version of the method, returns a new `ObserverMethodConfigurator` as defined in `ObserverMethodConfigurator` interface to easily configure the `ObserverMethod` which will be added at the end of observer invocation. If the container is unable to process the configurator it automatically detects the problem and treats it as a deployment problem.

- `addContext()` registers a custom `Context` object with the container.
- `getAnnotatedType()` and `getAnnotatedTypes()` returns the `AnnotatedType` s discovered or added

during container initialization. The id of an `AnnotatedType` added by the container is not defined. If the `id` passed is null, the container should substitute the container generated id.

A portable extension may take advantage of this event to register beans, interceptors, decorators, observer methods and custom context objects with the container.

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

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

If any `AfterBeanDiscovery` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

The container must execute the `@Synthesis` phase of build compatible extensions at this time.

23.6.3.1. `BeanConfigurator` interface

CDI 2.0 introduced the `jakarta.enterprise.inject.spi.configurator.BeanConfigurator` interface to help configuring a new `Bean` instance.

With `BeanConfigurator` you can perform all the operations defined in `BeanAttributesConfigurator` interface plus the following:

- Initialize the bean metadata with one of its `read()` methods. It can be done from an existing `BeanAttributes` or by reading metadata on a given `AnnotatedType`, according to the rules defined in Concepts.
- Set the class of the bean with `beanClass` method.
- Add an injection point for the bean with `addInjectionPoint` method.
- Add multiple injection points for the bean with `addInjectionPoints` methods.
- Replace all injection points for the bean with `injectionPoints` methods.
- Make the bean implements `PassivationCapable` and set its id with `id` method.
- Set the priority of the bean, if it is an alternative, with `priority` method.
- Set a callback to create a bean instance with `createWith()` or `produceWith()` method.
- Set a callback to destroy a bean instance with `destroyWith()` or `disposeWith()` method.

If a `BeanConfigurator` has no scope specified, the default scope rules, defined in Default scope, apply.

23.6.3.2. `ObserverMethodConfigurator` interface

CDI 2.0 introduced the `jakarta.enterprise.inject.spi.configurator.ObserverMethodConfigurator` interface to help configuring an `ObserverMethod` instance.

With `ObserverMethodConfigurator` you can perform the following operations:

- Read the observer metadata from a `java.lang.reflect.Method`, `AnnotatedMethod` or an existing `ObserverMethod` with one of its `read()` methods.
- Set the `ObserverMethod` bean class with `beanClass` method.
- Set the type of the observed event with `observedType` method.
- Add a qualifier with `addQualifier` method.
- Set or add multiple qualifiers with `addQualifiers` and `qualifiers` methods.
- Set the `Reception` type with reception method.
- Set the `TransactionPhase` type with `transactionPhase` method.
- Set the priority with `priority` method.
- Define the `EventConsumer` to call on notification with `notifyWith` method.
- Make the observer asynchronous with `async` method.

23.6.4. `AfterDeploymentValidation` event

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

The event object must be of type `jakarta.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.

If any `AfterDeploymentValidation` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

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

The container must execute the `@Validation` phase of build compatible extensions at this time.

23.6.5. `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 `jakarta.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.

23.6.6. `ProcessAnnotatedType` event

The container must fire an event, before it processes a type, for every Java class, interface (excluding *annotation type*, a special kind of interface type) or enum discovered as defined in Type discovery in CDI Full.

An event is not fired for any type annotated with `@Vetoed`, or in a package annotated with `@Vetoed`.

The event object must be of type `jakarta.enterprise.inject.spi.ProcessAnnotatedType<X>`, where `X` is the class, for types discovered in a bean archive, or of type `jakarta.enterprise.inject.spi.ProcessSyntheticAnnotatedType<X>` for types added by `BeforeBeanDiscovery.addAnnotatedType()` or `AfterTypeDiscovery.addAnnotatedType()`.

The annotation `@WithAnnotations` may be applied to the event parameter. If the annotation is applied, the container must only deliver `ProcessAnnotatedType` events for types which contain at least one of the annotations specified. The annotation can appear on the annotated type, or on any member, or any parameter of any member of the annotated type, as defined in Alternative metadata sources. The annotation may be applied as a meta-annotation on any annotation considered.

If the `@WithAnnotations` annotation is applied to any other event parameter, the container automatically detects the problem and treats it as a definition error.

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

```
interface ProcessSyntheticAnnotatedType<X> extends ProcessAnnotatedType<X> {
    public Extension getSource();
}
```

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

- `setAnnotatedType()` replaces the `AnnotatedType`.
- `configureAnnotatedType()` returns an `AnnotatedTypeConfigurator` (as defined in `AnnotatedTypeConfigurator` SPI) initialized with the `AnnotatedType` processed by the event to easily configure the `AnnotatedType` which will be used to replace the original one at the end of observer invocation. The method always returns the same `AnnotatedTypeConfigurator`
- `veto()` forces the container to ignore the type.
- `getSource()` returns the `Extension` instance that added the annotated type.

Any observer of this event is permitted to wrap and/or replace the `AnnotatedType` by calling either `setAnnotatedType()` or `configureAnnotatedType()`. If both methods are called within an observer notification an `IllegalStateException` is thrown. The container must use the final value of this property, after all observers have been called, as the only source of types and annotations for the program elements.

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

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

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

If any `ProcessAnnotatedType` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

The container must execute the `@Enhancement` phase of build compatible extensions at this time.

23.6.7. `ProcessInjectionPoint` event

The container must fire an event for every injection point of every bean, interceptor or decorator.

The event object must be of type `jakarta.enterprise.inject.spi.ProcessInjectionPoint<T, X>` where `T` is the bean class, and `X` is the declared type of the injection point.

```
public interface ProcessInjectionPoint<T, X> {
    public InjectionPoint getInjectionPoint();
    public void setInjectionPoint(InjectionPoint injectionPoint);
    public InjectionPointConfigurator configureInjectionPoint();
    public void addDefinitionError(Throwable t);
}
```

- `getInjectionPoint()` returns the `InjectionPoint` object that will be used by the container to perform injection.
- `setInjectionPoint()` replaces the `InjectionPoint`.

- `configureInjectionPoint()` returns an `InjectionPointConfigurator` (as defined in `InjectionPointConfigurator` interface) initialized with the `InjectionPoint` processed by the event to easily configure the `InjectionPoint` which will be used to replace the original one at the end of observer invocation. The method always returns the same `InjectionPointConfigurator`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `InjectionPoint` by calling either `setInjectionPoint()` or `configureInjectionPoint()`. If both methods are called within an observer notification an `IllegalStateException` is thrown. The container must use the final value of this property, after all observers have been called, whenever it performs injection upon the injection point.

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

If any `ProcessInjectionPoint` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

23.6.7.1. `InjectionPointConfigurator` interface

CDI 2.0 introduced the `jakarta.enterprise.inject.spi.configurator.InjectionPointConfigurator` interface to help configure an existing `InjectionPoint` instance.

With `InjectionPointConfigurator` you can perform the following operations:

- Set the type of `InjectionPoint` with `type` method.
- Add a qualifier with `addQualifier` method.
- Set or add multiple qualifiers with `addQualifiers` and `qualifiers` methods.
- Make the injection point delegate with `delegate` method.
- Make the injection point a transient field with `transientField` method.

23.6.8. `ProcessInjectionTarget` event

The container must fire an event for every bean, interceptor or decorator.

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

```
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 bean class.

- `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.

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

If any `ProcessInjectionTarget` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

23.6.9. `ProcessBeanAttributes` event

The container must fire an event for each managed bean, producer, interceptor or decorator deployed in a bean archive, before registering the `Bean` object. No event is fired for any:

- beans added programmatically using `AfterBeanDiscovery.addBean()`, or,
- for any built-in beans.

The event object must be of type `jakarta.enterprise.inject.spi.ProcessBeanAttributes<T>` where `T` is the bean class of the bean, the return type of the producer method, or the type of the producer field.

Resources are considered to be producer fields.

```
public interface ProcessBeanAttributes<T> {
    public Annotated getAnnotated();
    public BeanAttributes<T> getBeanAttributes();
    public void setBeanAttributes(BeanAttributes<T> beanAttributes);
    public BeanAttributesConfigurator<T> configureBeanAttributes();
    public void addDefinitionError(Throwable t);
    public void veto();
    public void ignoreFinalMethods();
}
```

- `getAnnotated()` returns the `AnnotatedType` representing the bean class, the `AnnotatedMethod` representing the producer field, or the `AnnotatedField` representing the producer field.
- `getBeanAttributes()` returns the `BeanAttributes` object that will be used by the container to manage instances of the bean.
- `setBeanAttributes()` replaces the `BeanAttributes`.
- `configureBeanAttributes()` returns a `BeanAttributesConfigurator` (as defined in `BeanAttributesConfigurator` interface) initialized with the `BeanAttributes` processed by the event

to easily configure the `BeanAttributes` which will be used to replace the original one at the end of observer invocation. The method always returns the same `BeanAttributesConfigurator`.

- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.
- `veto()` forces the container to ignore the bean.
- `ignoreFinalMethods()` Instructs the container to ignore all non-static, final methods with public, protected or default visibility declared on any bean type of the specific bean during validation of injection points that require proxyable bean type. These method should never be invoked upon bean instances. Otherwise, unpredictable behavior results. It will bypass standard rules defined in Unproxyable bean types.

Any observer of this event is permitted to wrap and/or replace the `BeanAttributes` by calling either `setBeanAttributes()` or `configureBeanAttributes()`. If both methods are called within an observer notification an `IllegalStateException` is thrown. The container must use the final value of this property, after all observers have been called, to manage instances of the bean. Changes to `BeanAttributes` are *not* propagated to the annotated type from which the bean definition was created.

Any bean which has its bean attributes altered must have it's definition validated during deployment validation.

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

If any `ProcessBeanAttributes` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

23.6.9.1. `BeanAttributesConfigurator` interface

CDI 2.0 introduced the `jakarta.enterprise.inject.spi.configurator.BeanAttributesConfigurator` interface to help configuring a new `BeanAttributes` instance.

`BeanAttributesConfigurator` is obtainable during `ProcessBeanAttributes` event and is therefore automatically initialized from existing `BeanAttributes`.

With `BeanAttributesConfigurator` you can perform the following operations :

- Add type with `addType` or `addTransitiveTypeClosure` methods.
- Set multiple types with `types` methods.
- Set scope with `scope` method.
- Add a qualifier with `addQualifier` method.
- Set or add multiple qualifiers with `addQualifiers` and `qualifiers` methods.
- Add a stereotype with `addStereotype` method.
- Set or add multiple stereotypes with `addStereotypes` and `stereotypes` methods.
- Set the bean name `name` method.

- Make the bean an alternative with `alternative` methods.

23.6.10. `ProcessBean` event

The container must fire an event for each bean, interceptor or decorator deployed in a bean archive, after firing the `ProcessBeanAttributes` for the bean and before registering the `Bean` object.

The event object type in the package `jakarta.enterprise.inject.spi` depends upon what kind of bean was discovered:

- For a managed bean with bean class `X`, the container must raise an event of type `ProcessManagedBean<X>`.
- For a producer method with method return type `T` of a bean with bean class `X`, the container must raise an event of type `ProcessProducerMethod<T, X>`.
- For a producer field with field type `T` of a bean with bean class `X`, the container must raise an event of type `ProcessProducerField<T, X>`.
- For a custom implementation of `Bean`, the container must raise an event of type `ProcessSyntheticBean<X>`.

The interface `jakarta.enterprise.inject.spi.ProcessBean` is a supertype of all these event types:

```
public interface ProcessBean<X> {
    public Annotated getAnnotated();
    public Bean<X> getBean();
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotated()` returns the `AnnotatedType` representing the bean class, the `AnnotatedMethod` representing the producer method, or the `AnnotatedField` representing the producer field. If invoked upon a `ProcessSyntheticBean` event, non-portable behavior results and the returned value should be ignored.
- `getBean()` returns the `Bean` object that is about to be registered. The `Bean` may implement `Interceptor` or `Decorator`.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

```
public interface ProcessManagedBean<X>
        extends ProcessBean<X> {
    public AnnotatedType<X> getAnnotatedBeanClass();
    public InvokerBuilder<Invoker<X, ?>> createInvoker(AnnotatedMethod<? super X>
method);
}
```

The `createInvoker` method allows creating an `InvokerBuilder` for the processed bean and the given target method (see Using `InvokerBuilder` in CDI Full).

```
public interface ProcessProducerMethod<T, X>
        extends ProcessBean<X> {
    public AnnotatedMethod<T> getAnnotatedProducerMethod();
    public AnnotatedParameter<T> getAnnotatedDisposedParameter();
}
```

```
public interface ProcessProducerField<T, X>
        extends ProcessBean<X> {
    public AnnotatedField<T> getAnnotatedProducerField();
    public AnnotatedParameter<T> getAnnotatedDisposedParameter();
}
```

```
public interface ProcessSyntheticBean<X>
        extends ProcessBean<X> {
    public Extension getSource();
}
```

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

If any `ProcessBean` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

The container must execute the bean-related part of `@Registration` phase of build compatible extensions at this time.

23.6.11. `ProcessProducer` event

The container must fire an event for each producer method or field of each bean, including resources.

The event object must be of type `jakarta.enterprise.inject.spi.ProcessProducer<T, X>`, where `T` is the bean class of the bean that declares the producer method or field and `X` is the return type of the producer method or the type of the producer field.

```
public interface ProcessProducer<T, X> {
    public AnnotatedMember<T> getAnnotatedMember();
    public Producer<X> getProducer();
    public void setProducer(Producer<X> producer);
    public ProducerConfigurator<X> configureProducer();
    public void addDefinitionError(Throwable t);
}
```

- `getAnnotatedMember()` returns the `AnnotatedField` representing the producer field or the `AnnotatedMethod` representing the producer method.

- `getProducer()` returns the `Producer` object that will be used by the container to call the producer method or read the producer field.
- `setProducer()` replaces the `Producer`.
- `configureProducer()` returns a `ProducerConfigurator` (as defined in `ProducerConfigurator` interface) initialized with the `Producer` processed by this event to configure the `Producer` that will replace the original one at the end of the observer invocation. Each call returns the same configurator instance within an observer notification.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.

Any observer of this event is permitted to wrap and/or replace the `Producer` by calling either `setProducer()` or `configureProducer()`. If both methods are called within an observer notification an `IllegalStateException` is thrown. The container must use the final value of this property, after all observers have been called, whenever it calls the producer or disposer.

For example, this observer decorates the `Producer` for all producer methods and fields of type `EntityManager`.

```
void decorateEntityManager(@Observes ProcessProducer<?, EntityManager> pp) {
    pit.setProducer( decorate( pp.getProducer() ) );
}
```

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

If any `ProcessProducer` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

23.6.11.1. `ProducerConfigurator` interface

CDI 2.0 introduced the `jakarta.enterprise.inject.spi.configurator.ProducerConfigurator` interface to help configuring a `Producer` instance.

With `ProducerConfigurator` you can perform the following operations:

- Set a callback to produce a new instance with `produceWith()` method.
- Set a callback to destroy the produced instance with `disposeWith()` method.

23.6.12. `ProcessObserverMethod` event

The container must fire an event for each observer method of each enabled bean, before registering the `ObserverMethod` object.

The event object must be of type `jakarta.enterprise.inject.spi.ProcessObserverMethod<T, X>`, where `T` is the observed event type of the observer method and `X` is the bean class of the bean that declares the observer method.

For a custom implementation of `jakarta.enterprise.inject.spi.ObserverMethod`, the container must raise an event of type `jakarta.enterprise.inject.spi.ProcessSyntheticObserverMethod<T, X>`, where `T` is the observed event type of the observer method and `X` is the return value of `ObserverMethod.getBeanClass()`.

```
public interface ProcessObserverMethod<T, X> {
    public AnnotatedMethod<X> getAnnotatedMethod();
    public ObserverMethod<T> getObserverMethod();
    public void addDefinitionError(Throwable t);
    public void setObserverMethod(ObserverMethod<T> observerMethod);
    public ObserverMethodConfigurator<T> setObserverMethod();
    public void veto();
}
```

```
public interface ProcessSyntheticObserverMethod<T, X> extends ProcessObserverMethod<T,
X> {
    public Extension getSource();
}
```

- `getAnnotatedMethod()` returns the `AnnotatedMethod` representing the observer method. If invoked upon a `ProcessSyntheticObserverMethod` event, non-portable behavior results and the returned value should be ignored.
- `getObserverMethod()` returns the `ObserverMethod` object that will be used by the container to call the observer method.
- `addDefinitionError()` registers a definition error with the container, causing the container to abort deployment after bean discovery is complete.
- `setObserverMethod()` replaces the `ObserverMethod`.
- `configureObserverMethod()` returns an `ObserverMethodConfigurator` (as defined in `ObserverMethodConfigurator` interface) initialized with the `ObserverMethod` processed by the event to easily configure the `ObserverMethod` which will be used to replace the original one at the end of observer invocation. The method always returns the same `ObserverMethodConfigurator`.
- `veto()` forces the container to ignore the `ObserverMethod`.
- `getSource()` returns the `Extension` instance that added the observer method.

Any observer of this event is permitted to wrap and/or replace the `ObserverMethod` by calling either `setObserverMethod()` or `configureObserverMethod()`. If both methods are called within an observer notification an `IllegalStateException` is thrown. The container must use the final value of this property, after all observers have been called, whenever it performs observer resolution.

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

If any `ProcessObserverMethod` method is called outside of the observer method invocation, an `IllegalStateException` is thrown.

The container must execute the observer-related part of `@Registration` phase of build compatible extensions at this time.

23.7. Configurators interfaces

CDI 2.0 introduced the following Configurators interface:

- `AnnotatedTypeConfigurator` SPI for `AnnotatedType` configuration
- `InjectionPointConfigurator` interface for `InjectionPoint` configuration
- `BeanAttributesConfigurator` interface for `BeanAttributes` configuration
- `BeanConfigurator` interface for `Bean` configuration
- `ObserverMethodConfigurator` interface for `ObserverMethod` configuration
- `ProducerConfigurator` interface for `Producer` configuration

The container must provide implementation for all these configurators and make them available in matching container lifecycle events as defined in Container lifecycle events.

23.8. The `InterceptionFactory` interface

CDI 2.0 introduces the `jakarta.enterprise.inject.spi.InterceptionFactory<T>` interface, which allows to create a wrapper instance whose method invocations are intercepted by method interceptors and forwarded to a provided instance.

```
public interface InterceptionFactory<T> {
    InterceptionFactory<T> ignoreFinalMethods();
    AnnotatedTypeConfigurator<T> configure();
    T createInterceptedInstance(T instance);
}
```

- `ignoreFinalMethods()` instructs the container to ignore all non-static, final methods with public, protected or default visibility declared by any class in the type hierarchy of the intercepted instance during invocation of `createInterceptedInstance()` method. Ignored methods should never be invoked upon the wrapper instance. Otherwise, unpredictable behavior results.
- `configure()` returns an `AnnotatedTypeConfigurator` (as defined in `AnnotatedTypeConfigurator` SPI) initialized with the `AnnotatedType` created either for the class passed to `BeanManager.createInterceptionFactory(CreationalContext, Class)` or derived from the `InterceptionFactory` parameter injection point. The method always return the same `AnnotatedTypeConfigurator`
- `createInterceptedInstance()` returns a wrapper instance whose method invocations are intercepted by method interceptors and forwarded to a provided instance. The method can be only called once, subsequent calls will throw an `IllegalStateException`. If the type of the instance is not proxyable as defined in Unproxyable bean types an `UnproxyableResolutionException` exception is thrown. This rule can be loosen by calling `ignoreFinalMethods()` before this method. If the provided instance is an internal container

construct (such as client proxy), non-portable behavior results.

An `InterceptionFactory` can be obtained be calling `BeanManager.createInterceptionFactory()` as defined in Obtain an `InterceptionFactory`.

The container must provide a built-in bean with scope `@Dependent`, bean type `InterceptionFactory` and qualifier `@Default`.

If an injection point of type `InterceptionFactory` and qualifier `@Default` exists and is not a parameter of a producer method, the container automatically detects the problem and treats it as a definition error.

If an injection point of type `InterceptionFactory` has a type parameter that is not a Java class, non-portable behavior results.

The following example demonstrates a producer method definition using `InterceptionFactory`. The produced bean instance will be a wrapper of `Product` with single interceptor associated by `ActionBinding`:

```
@Produces
@RequestScoped
public Product createInterceptedProduct(InterceptionFactory<Product>
interceptionFactory) {
  interceptionFactory.configure().add(ActionBinding.Literal.INSTANCE);
  return interceptionFactory.createInterceptedInstance(new Product());
}
```

Chapter 24. Packaging and deployment in CDI Full

This chapter replaces Packaging and deployment for the purpose of CDI Full. The Packaging and deployment chapter should be considered merely informative. In CDI Full, the term *deployment time* always means during application startup.

When an application is started, the container must perform *bean discovery*, detect definition errors and deployment problems and raise events that allow portable extensions to integrate with the deployment lifecycle.

Bean discovery is the process of determining:

- The bean archives that exist in the application, and the beans they contain
- Which alternatives, interceptors and decorators are *enabled* for each bean archive
- The *ordering* of enabled interceptors and decorators

Additional beans may be registered programmatically with the container by the application or a portable extension after the automatic bean discovery completes. Portable extensions may even integrate with the process of building the `Bean` object for a bean, to enhance the container's built-in functionality.

24.1. Bean archives in CDI Full

Bean classes of enabled beans must be deployed in *bean archives*.

A bean archive has a *bean discovery mode* of `all`, `annotated` or `none`. A bean archive which contains non-empty `beans.xml` must specify the `bean-discovery-mode` attribute. The default value for the attribute is `annotated`.

An archive which:

- contains a `beans.xml` file with the `bean-discovery-mode` of `none`, or,
- contains a portable extension or a build compatible extension and no `beans.xml` file

is not a bean archive.

An *explicit bean archive* is an archive which contains a `beans.xml` file with `bean-discovery-mode` of `all`.

An *implicit bean archive* is:

- an archive which contains a `beans.xml` file that is empty, or,
- any other archive which contains one or more bean classes with a bean defining annotation as defined in Bean defining annotations.

When determining which archives are bean archives, the container must consider:

- Library jars
- Directories in the JVM classpath

Non Jakarta EE containers may or may not provide support for war, EJB jar or rar bean archives.

The `beans.xml` file must be named:

- `META-INF/beans.xml`.

For compatibility with CDI versions prior to 4.0, CDI Full products must contain an option that causes an archive with empty beans.xml to be considered an explicit bean archive.

The container searches for beans in all bean archives in the application classpath.

If a bean class is deployed in two different bean archives, non-portable behavior results. Portable applications must deploy each bean class in no more than one bean archive.

Explicit bean archives may contain classes which are not deployed as beans. For example a bean archive might contain an abstract class not annotated with `@Decorator`.

Implicit bean archives are likely to contain classes which are not deployed as beans.

An extension may be deployed in any archive, including those that are not bean archives.

24.2. Application initialization lifecycle in CDI Full

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

- First, the container must search for service providers for the service `jakarta.enterprise.inject.spi.Extension` defined in 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 `BeforeBeanDiscovery` event.
- Next, the container must perform type discovery, as defined in Type discovery in CDI Full.
- Next, the container must fire an event of type `AfterTypeDiscovery`, as defined in `AfterTypeDiscovery` event.
- Next, the container must perform bean discovery, as defined in Bean discovery in CDI Full.
- Next, the container must fire an event of type `AfterBeanDiscovery`, as defined in `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, specialization and invoker lookups and abort initialization of the application if any deployment problems exist, as defined in Problems detected automatically by the container.
- Next, the container must fire an event of type `AfterDeploymentValidation`, as defined in `AfterDeploymentValidation` event, and abort initialization of the application if any observer registers a deployment problem.

- Finally, the container begins directing requests to the application.

24.3. Application shutdown lifecycle in CDI Full

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

- First, the container must destroy all contexts.
- Finally, the container must fire an event of type `BeforeShutdown`, as defined in `BeforeShutdown` event.

24.4. Type and Bean discovery in CDI Full

The container automatically discovers managed beans (according to the rules of Which Java classes are managed beans?) in bean archives and searches the bean classes for producer methods, producer fields, disposer methods and observer methods.

24.4.1. Type discovery in CDI Full

First the container must discover types. The container discovers:

- each Java class, interface (excluding the special kind of interface declaration *annotation type*) or enum deployed in an explicit bean archive, and
- each Java class with a bean defining annotation in an implicit bean archive,

that is not excluded from discovery by an *exclude filter* as defined in Exclude filters.

Then, for every type discovered the container must create an `AnnotatedType` representing the type and fire an event of type `ProcessAnnotatedType`, as defined in `ProcessAnnotatedType` event.

If an extension calls `BeforeBeanDiscovery.addAnnotatedType()` or `AfterTypeDiscovery.addAnnotatedType()`, the type passed must be added to the set of discovered types and the container must fire an event of type `ProcessSyntheticAnnotatedType` for every type added, as defined in `ProcessAnnotatedType` event+

24.4.2. Exclude filters

Exclude filters are defined by `<exclude>` elements in the `beans.xml` for the bean archive as children of the `<scan>` element. By default an exclude filter is active. If the exclude filter definition contains:

- a child element named `<if-class-available>` with a `name` attribute, and the classloader for the bean archive can not load a class for that name, or
- a child element named `<if-class-not-available>` with a `name` attribute, and the classloader for the bean archive can load a class for that name, or
- a child element named `<if-system-property>` with a `name` attribute, and there is no system property defined for that name, or
- a child element named `<if-system-property>` with a `name` attribute and a `value` attribute, and there is no system property defined for that name with that value.

then the filter is inactive.

If the filter is active, and:

- the fully qualified name of the type being discovered matches the value of the name attribute of the exclude filter, or
- the package name of the type being discovered matches the value of the name attribute with a suffix ".*" of the exclude filter, or
- the package name of the type being discovered starts with the value of the name attribute with a suffix ".**" of the exclude filter

then we say that the type is excluded from discovery.

For example, consider the follow `beans.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
       bean-discovery-mode="all" version="3.0">

    <scan>
        <exclude name="com.acme.rest.*" />

        <exclude name="com.acme.faces.**">
            <if-class-not-available name="jakarta.faces.context.FacesContext"/>
        </exclude>

        <exclude name="com.acme.verbose.*">
            <if-system-property name="verbosity" value="low"/>
        </exclude>

        <exclude name="com.acme.ejb.**">
            <if-class-available name="jakarta.enterprise.inject.Model"/>
            <if-system-property name="exclude-ejbs"/>
        </exclude>
    </scan>

</beans>
```

The first exclude filter will exclude all classes in `com.acme.rest` package. The second exclude filter will exclude all classes in the `com.acme.faces` package, and any subpackages, but only if JSF is not available. The third exclude filter will exclude all classes in the `com.acme.verbose` package if the system property `verbosity` has the value `low`. The fourth exclude filter will exclude all classes in the `com.acme.ejb` package, and any subpackages if the system property `exclude-ejbs` is set (with any value) and at the same time, the `jakarta.enterprise.inject.Model` class is available to the classloader.

24.4.3. Trimmed bean archive

An explicit bean archive may be marked as 'trimmed' by adding the `<trim />` element to its `beans.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
        version="3.0">

    <trim/>
</beans>
```

If an explicit bean archive contains the `<trim/>` element in its `beans.xml` file, types that don't have either a bean defining annotation (as defined in Bean defining annotations) or any scope annotation, are removed from the set of discovered types.

24.4.4. Bean discovery in CDI Full

For every type in the set of discovered types (as defined in Type discovery in CDI Full), the container must:

- inspect the type metadata to determine if it is a bean, and then
- detect definition errors by validating the class and its metadata, and then
- if the class is a managed bean, fire an event of type `ProcessInjectionPoint` for each injection point in the class, as defined in `ProcessInjectionPoint` event, and then
- if the class is a managed bean, fire an event of type `ProcessInjectionTarget`, as defined in `ProcessInjectionTarget` event, and then
- determine which alternatives, interceptors and decorators are enabled, according to the rules defined in Enabled and disabled beans, Interceptor enablement and ordering in CDI Full and Decorator enablement and ordering, and then
- if the class is an enabled bean, interceptor or decorator, fire an event of type `ProcessBeanAttributes`, as defined in `ProcessBeanAttributes` event, and then
- if the class is an enabled bean, interceptor or decorator and if `ProcessBeanAttributes.veto()` wasn't called in previous step, fire an event which is a subtype of `ProcessBean`, as defined in `ProcessBean` event.

For each enabled bean, the container must search the class for producer methods and fields, as defined in Producer methods and in Producer fields, including resources, and for each producer:

- if it is a producer method, fire an event of type `ProcessInjectionPoint` for each injection point in the method parameters, as defined in `ProcessInjectionPoint` event, and then
- fire an event of type `ProcessProducer`, as defined in `ProcessProducer` event, and then

- if the producer method or field is enabled, fire an event of type `ProcessBeanAttributes`, as defined in `ProcessBeanAttributes` event, and then
- if the producer method or field is enabled and if `ProcessBeanAttributes.veto()` wasn't called in previous step, fire an event which is a subtype of `ProcessBean`, as defined in `ProcessBean` event.

For each enabled bean, the container must search for disposer methods as defined in Disposer methods, and for each disposer method:

- fire an event of type `ProcessInjectionPoint` for each injection point in the method parameters, as defined in `ProcessInjectionPoint` event.

For each enabled bean, the container must search the class for observer methods, and for each observer method:

- fire an event of type `ProcessInjectionPoint` for each injection point in the method parameters, as defined in `ProcessInjectionPoint` event, and then
- fire an event of type `ProcessObserverMethod`, as defined in `ProcessObserverMethod` event.

Then, the container registers the `Bean` and `ObserverMethod` objects:

- For each enabled bean that is not an interceptor or decorator, the container registers an instance of the `Bean` interface defined in The `Bean` interface.
- For each enabled interceptor, the container registers an instance of the `Interceptor` interface defined in The `Interceptor` interface.
- For each enabled decorator, the container registers an instance of the `Decorator` interface defined in The `Decorator` interface.
- For each observer method of every enabled bean, the container registers an instance of the `ObserverMethod` interface defined in The `ObserverMethod` interface.

Part II - CDI in Java SE

This part of the document specifies additional rules and features when using CDI in Java SE. All content defined in Part I - Core CDI applies to this part.

CDI implementations that support the Java SE API are required to support CDI Full.

Chapter 25. Bootstrapping a CDI container in Java SE

In Java SE, the CDI container must be explicitly bootstrapped by the user. This is performed by the `SeContainerInitializer` abstract class and its static method `newInstance()`.

`SeContainerInitializer` is a service provider of the service `jakarta.enterprise.inject.se.SeContainerInitializer` declared in META-INF/services. This class allows a user to configure the CDI container before it is bootstrapped. The `SeContainerInitializer.initialize()` method bootstraps the container and returns a `SeContainer` instance.

User can shutdown the container manually by calling the `close()` method on `SeContainer` or automatically using try-with-resources since `SeContainer` extends `AutoCloseable` interface.

25.1. `SeContainerInitializer` class

CDI container can be configured and bootstrapped from `jakarta.enterprise.inject.se.SeContainerInitializer` abstract class.

A CDI implementation is required to provide an implementation of `SeContainerInitializer` declared as a service provider. Static method `newInstance()` uses Java service provider mechanism to obtain an implementation of `SeContainerInitializer` and return an instance of it. There must be exactly one provider available, otherwise an `IllegalStateException` is thrown.

`SeContainerInitializer` configuration allows explicit addition of elements to the set of automatically discovered elements. These additions are done in an internal synthetic bean archive that is added to the set of bean archives discovered by the container during deployment.

This synthetic bean archive behaves like an explicit bean archive (as defined in Bean archives in CDI Full).

```
public abstract class SeContainerInitializer {
    public static SeContainerInitializer newInstance() { ... }
    public SeContainerInitializer addBeanClasses(Class<?>... classes);
    public SeContainerInitializer addPackages(Class<?>... packageClasses);
    public SeContainerInitializer addPackages(boolean scanRecursively, Class<?>...
packageClasses);
    public SeContainerInitializer addPackages(Package... packages);
    public SeContainerInitializer addPackages(boolean scanRecursively, Package...
packages);
    public SeContainerInitializer addExtensions(Extension... extensions);
    public SeContainerInitializer addExtensions(Class<? extends Extension>...
extensions);
    public SeContainerInitializer enableInterceptors(Class<?>... interceptorClasses);
    public SeContainerInitializer enableDecorators(Class<?>... decoratorClasses);
    public SeContainerInitializer selectAlternatives(Class<?>... alternativeClasses);
    public SeContainerInitializer selectAlternativeStereotypes(Class<? extends
```

```
Annotation>... alternativeStereotypeClasses);
    public SeContainerInitializer addProperty(String key, Object value);
    public SeContainerInitializer setProperties(Map<String, Object> properties);
    public SeContainerInitializer disableDiscovery();
    public SeContainerInitializer setClassLoader(ClassLoader classLoader);
    public SeContainer initialize();
}
```

Unless specified differently each method of `SeContainerInitializer` returns the current `SeContainerInitializer` object.

- `newInstance()` static method returns an instance of the implementation of `SeContainerInitializer` discovered by Java service provider. Each call returns a new instance of `SeContainerInitializer`. This method throws `IllegalStateException` if called in Jakarta EE container.
- `addBeanClasses()` adds classes to the the synthetic bean archive
- `addPackages()` adds packages content to the synthetic bean archive. There are other versions of this method, which enables user to add a package according to class or classes it contains and also to add packages recursively.
- `addExtensions()` adds the provided extensions (class or instance) to the synthetic bean archive.
- `enableInterceptors()` adds interceptor classes to the list of enabled interceptors for the synthetic bean archive.
- `enableDecorators()` adds decorator classes to the list of enabled decorators for the synthetic bean archive.
- `selectAlternatives()` adds alternatives classes to the list of selected alternatives for the synthetic bean archive.
- `selectAlternativeStereotypes()` adds alternative stereotype classes to the list of selected alternative stereotypes for the synthetic bean archive.
- `addProperty()` adds a configuration property to the container
- `setProperties()` sets the `Map` of configuration properties for the container. Original properties `Map` is replaced by a new instance.
- `disableDiscovery()` deactivates automatic type scanning and discovery. All bean archives will be ignored except the implicit bean archive.
- `setClassLoader()` changes the default class loader for the container
- `initialize()` bootstraps the container and returns a `SeContainer` as defined in `SeContainer` interface.

Every invocation of the `SeContainerInitializer.initialize()` method returns a new `SeContainer` instance. The application context is started automatically by the container on start up. An implementation does not need to support multiple calls of `SeContainerInitializer.initialize()` method when the `SeContainer` is running.

25.2. SeContainer interface

The jakarta.enterprise.inject.se.SeContainer interface provides access to the BeanManager and programmatic lookup as defined in The Instance interface. SeContainer also extends AutoCloseable, so when dereferenced, it should shutdown automatically.

```
public interface SeContainer extends Instance<Object>,AutoCloseable {
    void close();
    boolean isRunning();
    BeanManager getBeanManager();
}
```

- close() method explicitly shuts down the container. If it is called and the container was already shutdown, it throws an IllegalStateException.
- isRunning() method returns true if called before container shuts down and false after.
- getBeanManager() method returns the BeanManager (as defined in The BeanManager object) for the running container. If it is called and the container was already shutdown, it throws an IllegalStateException.

SeContainer implements jakarta.enterprise.inject.Instance and therefore might be used to perform programmatic lookup as defined in The Instance interface. If no qualifier is passed to SeContainer.select() method, the @Default qualifier is assumed.

If any Instance.select() method is called and the container was already shutdown, the IllegalStateException is thrown.

The following code examples demonstrate the options of handling container shutdown:

```
public static void main(String... args) {
    SeContainerInitializer containerInit = SeContainerInitializer.newInstance();
    SeContainer container = containerInit.initialize();
    // retrieve a bean and do work with it
    MyBean myBean = container.select(MyBean.class).get();
    myBean.doWork();
    // when done
    container.close();
}
```

```
public static void main(String... args) {
    try(SeContainer container = SeContainerInitializer.newInstance().initialize()) {
        // start the container, retrieve a bean and do work with it
        MyBean myBean = container.select(MyBean.class).get();
        myBean.doWork();
    }
    // shuts down automatically after the try with resources block.
}
```

Chapter 26. Scopes and contexts in Java SE

26.1. Context management for built-in scopes in Java SE

When running in Java SE, the container must extend the rules defined in Context management for built-in scopes and is also required to ensure the following rules for built-in context implementation.

26.1.1. Application context lifecycle in Java SE

When running in Java SE the container must extend the rules defined in Application context lifecycle and is also required to ensure the following rules.

The application scope is active:

- during any method invocation

The application context is shared between all method invocations that execute within the same container.

The application context is destroyed when the container is shut down.

The payload of the event fired when the application context is initialized or destroyed is:

- any `java.lang.Object`.

Chapter 27. Packaging and deployment in Java SE

27.1. Bean archive in Java SE

When running in Java SE, the container must extend the rules defined in Bean archives in CDI Full and also ensure that :

An archive which doesn't contain a `beans.xml` file can't be discovered as an *implicit bean archive* unless:

- the application is launched with system property `jakarta.enterprise.inject.scan.implicit` set to `true`, or
- the container was initialized with a map containing an entry parameter with `jakarta.enterprise.inject.scan.implicit` as key and `Boolean.TRUE` as value.

Chapter 28. Portable extensions in Java SE

28.1. The `BeanManager` object in Java SE

28.1.1. Obtaining a reference to the CDI container in Java SE

In Java SE, while the access to CDI container and `BeanManager` described in Obtaining a reference to the CDI container is available, the preferred way to access them is through `SeContainer` interface as described in `SeContainer` interface.