CDI

# What is the goal?

Java as an Object oriented language allows to define object classes which then can have any number of instances in runtime. A class defines the methods that another class can use to perform a given task and at the same time defines its internal datastructures and private help methods to undertake the file public helper class.

A class at is own is a compile-time artifact. In runtime classes aren’t used directlty. What exists, is a set of instances from a class or objects. Each instance is created having the same methods as its class but having its own values for the class datastructures which are stored in a diferent memory zone from the other class instances.

The java language does not impose a restriction to the number of instances a class migth have (except for the special class type enum).

A class instance (or class objects) are created using the **new** operator together with a constructor method defined by the class. The new operator reserves a memory zone for the fields values. And returns a reference to it. Any other class instance that wants to interact with the other class instance, must have access to its reference and call the public methods or public fields define the the instance class.

In big programs it is common that a instance class calls another object from another class whitch then create a new instance and then a third object. This type of dependecy is perfect normal in JAVA. But when employed with unit testing, where each class method results is tested given a set of input values, it is dificult to define tests because to test a class it is also necessary to test the classes that it dependes.

The solution for the problem is not allow the creation of instances inside another instance but give (or inject) references from other instances when an instance is created. This is called dependency injection.

The Java language does not provide any special support to implement dependency injection pattern. However it can be implemented by never creating objects inside another object but passing references to the needed objects using constructors or set methods.

In order to have a more formal dependency injection implementation to the java world, the CDI specification was developed.

# CDI

CDI is an specification of framework developed by the java comunity to enable dependency injection in a more formal way. The JDK does not implement this specification. To used it, is necessary to employ an external library, like for instance Weld.

The main concept of CDI is the programmer don’t explicit create class instance objects using the new operator, instead declares to CDI:

* what are the classes that can be created by CDI
* where it wants to be those instances

Every CDI functionality is implemented by a CDI container. How an application interacts with the CDI container is defined by the CDI API. A CDI container can run on its own Java SE process or can be integrated with a JAVA EE web-container or Java EE enterprise java beans.

A CDI container works with what is called beans. Beans are java classes that follow the CDI rules. The CDI engine creates Beans instances and stored them on a context. A context is a memory zone that groups all beans which have the same context type. When a context needs to be deleted then all instance beans are also destroyed. A Container instance can have several contexts simultaneous. Predefined it has the Application, Session (all instances live for the user session time), request, conversation.

A CDI container has the following responsabilities:

* Discover what beans are defined on a application (it analyses at startup time all jars with the goal to find out the classes of beans that be a source of instances) ;
* In runtime to discovers the places where the instance beans are needed to be available : the injection point
* It provides to the injection injection point the correct class instance. For doing this, the CDI container searchs all the application scopes for a maching instance or creates a new one if not found. Bare in mind that each class can define also injections point that are needed to fullfil when in order the instance to be created.

# How to define beans?

A CDI bean can be defined by :

* + A pojo class
  + A method on a bean
  + A property on a bean
  + EJB (not taken in consideration for this document)
  + Resources (java EE resources, persistence units, and web services) (not taken in consideration for this document)

## A pojo class

It its a java class with the following properties:

* Regarding the constructor, if:
  + Has no constructor (use the default) or;
  + Has the no argument constructor or;
  + Has just one constructor with arguments, but this constructor is annotated with @Inject
* It cannot be a non static- inner class
* It must be a concrete class
* An not configured as an EJB component

## Producer Method

A producer method is a method from a CDI bean that returns (or produces) when is called by the CDI Container a new bean instance. Its return type defines the type that will be stored on the context.

What is needed to define a producer method:

1. Be a method from a CDI Bean
2. The method have the annotation @javax.enterprise.inject.produces
3. Be a non-abstract method, static/ non-static, pulich/private/protected
4. Specify a scope for the returned Bean. If the producer method can return null then it must have the scope @Dependent
5. The return type can be:
   1. a Java class
   2. Primitive type
   3. An array
6. The method can have parameters. When the method is called, the parameter values are provided by the CDI Container (they not need to be explicit annotated with the injected annotation).

A CDI bean can have multiple producer methods.

## A producer field

It is a field declared on a CDI Bean. Its value must be given by the CDI Bean it self before it is called to CDI Container.

What is needed to define a producer field:

1. Be defined on a CDI Bean
2. The method have the annotation @javax.enterprise.inject.produces
3. Might be a static/non-static private/public/protected field
4. Must a scope specified
5. If it can return a null value it must be specified @Dependent scope
6. The field type can be:
   1. Java class
   2. Interface
   3. Primitive type
   4. Array type

A bean may declare multiple producer fields.

## Disposable method

An CDI Bean instance provided to the container by an producer method or field before it is deleted from memory can be given to a special method, where clean up operations using the instance could be performed.

How to declare

* Create a method on a CDI Bean
* The method can be a static/non-static public/private/protected/default-access
* It must have one or more parameters, where one and only one must be annotated with @javax.enterprise.inject.Disposes. The other parameters are injected by the CDI Container.
* The disposer method must be declared on the same bean as the producer method/field that created the CDI Bean instance that is disposed be the disposed method
* The disposes parameter must be of the same type as the producer type.

On a servelt container the CDI Container provide the following beans:

* HttpServletRequest
* HttpSession
* ServletContext

## injection points order while creating a bean

When a pojo CDI Bean instance is created, the CDI Container inject with other beans instances on the following points when the bean instance is initialized. With :

1. The constructor annotated by @javax.inject.Inject (if no constructor is defined with the injected annotation the the no argument constructor is called instead)
2. Fields annotated with the annotation @inject
   * + - Non-static
3. Zero or more Initialization methods:
   * + - Non-static, public, protected, private, default-access methods annotated by @injected
4. The @PostConstructor methods are called after all injection points have been resolved with the matching bean instance.

When an injection point does not declare an qualifier annotation then it is assumed the @Default qualifier.

A bean class can have a @PostDestroy annotated method that is called by the container when an instance is destroyed. It is garantee only after the execution end of this method the dependend scope instances are destroyed.

## The @New qualifier

The @New(type class) qualifier was deprecated on CDI 1.1 . Instead in should be used @Dependent scoped beans.

The new qualifier is a way to create a new @Dependent instance from an Application, session scope.

### Bean types

A bean class defines a type for the beans that an be instanciated from. In fact, a bean class can defines one or more types. The bean types can be found by the container in two ways:

* The analying the bean class declaration. In this case the types are the reunion of:
* The class of the bean it self;
* Every class enhirent by its parent class
* Every interface implemented by the bean class
* The java.lang.Object
* Reading the types the programmer has specified using the annotation @javax.enterprise.inject.Typed. In this case the type of the bean class is:
  + The type present on the Typed annotation parameter
  + The java.lang.Object

### Contexts

The predefined context are:

* ApplicationScope
* RequestScope
* SessionScope
* ConversationScope
* Dependent

A contextual reference to a bean with a normal scope, is not a direct reference to a contextual instance of a bean. Instead, the contextual reference is a client proxy object. A client proxy implements/extends some or all of the bean types of bean and delegates all methods calls to the current instance.

On a client proxy the methods from java.lang.Object should not be invoked except the toString() method.

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

### Chapter 5

The container injects references on:

* 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

Bean instance references can be also obtained by programmatic lookup or by expression language (not cover in this document).

When resolving a bean at an injection point, the container considers:

* Bean type
* Qualifiers
* Selected alternatives

# Modularity

For the beans to be discoverd by the contaainer they must be inside a bean module. A bean module is a Java EE module that have a file with the name bean.xml. A java EE module can be a WAR, EAR or JAR file.

# Alternatives

An alternative is a bean that a container can instanciate in place of another bean with the same type and qualifier. The alterative bean usualy has a diferent method implementation. Its main use case is in helping performing testcases.

There are two kind of alternatives:

* Alternatives for an application : Since CDI 1.1 is possible to specify an alternative bean that is not present on the same bean archive which defines the original bean.
* Alternatives for a bean Archive: the alternative bean is an altenative for a bean exclusive defined on the same archive bean module.

### How to define an alternative

First the alternate bean class must be annotated with @Alternative. Then depending from the alternative type:

* If it is a alternative for the application then it is only necessary to :
  + Place the @Priority annotation on the bean class
  + Place the @Priority annotation on the bean class that declares the producer method or producer field

The alternative bean with the highest priority is the one to be selected as an alternative for an application.

* If it is a alternative for a bean archive then the bean.xml file must be field with the tag alternatives. This new tag identifies
  + the bean classes that are the alternative or the bean classes that declare a producer field or method. This is done by using the using the sub tag class
  + or the annotation classes that are an @Alternative stereotype. This is done by using the sub tag stereotypes.

An alternative is detected and selected during the container startup procedure and continues active until the container stops. It is not possible to change an alternative bean while the container is running.

# Typesafe resolution

The process of finding a bean for an injection point is called typesafe resolution. A CDI container does the typesafe resolution procedure only one time during the application initialization time. If it is found any problem during this procedure the application is not started and the programmer is warned about it.

The container reads the required bean type and the required qualifiers present on the injection point and tries to find a declared bean with a type and qualifiers compatible with the injection point according to the following rules:

* The bean has a type that matches the required type. For this purpose, primitive types are considered to match their corresponding wrapper types in java.lang and arrays types are considered to match only if their element types are identical.
* The bean has all 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 :
  + The same type and
  + The same annotation member value for each member which is not annotated with @java.enterprise.util.Nonbinding.

Using the required bean type and required qualifiers list of an injection point the container finds, using the rules above, one and only one declared bean for the injection. If it can find no one then the injection point has an **unsatisfied dependency**. It it can find more than one than the injection point has an **ambiguous dependency**.

When dealing with primitive types both the primitive type and the corresponding wrapper classes are considered the same type. The container will automatically perform the boxing and unboxing when a value is injected. If a producer method or field with a primitive type returns null, then the injection point with the primitive type will be filled with the type default value.

# Qualifiers

Qualifiers are a mechanism to resolve the ambiguous dependency issue from an injection point. When two or more bean classes have the save type then it is possible to annotate one of the class beans with an Qualifier annotation to diferenciate it from the others. Then the injection point must also specified the qualifier annotation to have an instance from it.

There are two types of qualifiers:

* Predefined by the CDI specification:
  + @Default : Any bean class or injection point that does not specify any qualifier will have by default this qualifier.
  + @Name : special annotation that specified the bean class name on a JSF EL. (not cover here)
  + @New : special qualifier used on injection points that need a dependent instance
  + @Any : Every bean class will have this qualifier automatically declared.
* Programmer defined, A new annotation with the following rules:
  + Must be annotated with @javax.inject.Qualifier
  + @Retention(RUNTIME)
  + @Target({METHOD, FIELD, PARAMETER, TYPE})
  + It can have zero or more members .
    - To match the qualifiers defined on the injection point and on the bean classes the values of each annotation member are also compared using the equals() method. To exclude an annotation member from this comparison the member can be annotated with @NonBinding annotation.

# Injection point metadata

The bean class or a producer method can ask the CDI container for information about the injection point where an instance is required to be injected. To know this, a bean class can ask for the injection on one of its fields or parameters from a data initialization method or constructor for the **javax.enterprise.inject.spi.InjectionPoint** type. This type can also be injected as a producer method parameter.

The **InjectionPoint** type has the following members:

|  |  |
| --- | --- |
| public Type getType(); | returns the required type on the injection point; |
| public Set<Annotation> getQualifiers(); | returns the required Qualifiers on the injection point; |
| public Bean<?> getBean(); | returns the Bean object that represents the injection point; |
| public Member getMember(); | Returns the reflection type for the field, method or constructor where the injection point belongs |
| public Annotated getAnnotated(); | returns an javax.enterprise.inject.spi.AnnotatedField or javax.enterprise.spi.AnnotatedParameter, is the injection point is a field or a method/constructor parameter. |
| public boolean isDelegate(); | returns true if the injection point is a decorator delegate injection point |
| public boolean isTransient(); | returns true if the injection point id a transient field. |

Any scope other than Dependent must have an Injection point of type InjectionPoint with specific qualifier(s) declated. If declares no qualifier or the @Default then the container will throw an error. This rule does not apply to the @Dependent scope. In this scope it is possible to have use the @Default qualifier with a InjectionPoint type.

# Programatic lookup

A Dependent scope bean of javax.enterprise.inject.Instance<T> type allows to a programer to interact with the CDI container in order to programmatically retrieve an contextualized instance of a bean of type T, simulating what the @Inject annotation does. If the injection point where the bean Inject type is going to be injected has a qualifiers list than the Instance type will only work with instances that are of type T and declare the required qualifiers: exemple:

@Inject @Check Instance<PaymentProc> chequesPayProc;

The Instance interface is defined as:

Public interface Instance<T> extends Iterable<T>, Provider<T> {

Public Instance<T> select(Annotation... qualifiers); (1)

Public <U extends T> Instance<U> select(Class<U> subType, Annotation ... qualifiers); (2)

Public <U extends T> Instance<U> select(TypeLiteral<U> subType, Annotation.. qualifiers); (3)

Public boolean isUnsatisfied(); (4)

Public boolean isAmbiguous(); (5)

Public void destroy(T instance); (6)

}

1) 2) 3) Allows to access bean instances with subtype or qualifiers from the original Intance type. 3) is used to specify parameterized type with actual type parameters.

1. tells if the CDI container has not found any instances from the Bean with the same type and qualifiers as the Instance type.
2. tells if the CDI container has found more than one instances from the Bean with the same type and qualifiers as the Instance type.
3. Instructs the CDI to delete an bean instance. It should be a dependent scoped bean instance or a client proxy for a normal scoped bean.

The Provider interface adds a method get() that returns an instance from the bean with the type and qualifiers associated with the inject Instance type.

The Iterable interface adds a method iterator() that returns an iterator to all existing bean instance with the same type and qualifieres of Instance type injected instance.

# Decorators

A decorator implements one or more bean types and intercepts business methods invocation of beans which implement those beans types. These are called the decorated types.

A decorator allows to add extra functionality to other CDI beans, as long as they share a bean type. The CDI container intercepts the original decorated type methods invocations and first calls the matching method from the registered decorators, afterwards the method from the decorated type is invocated.

A decorator instance is a dependent object of the object it decorates.

## What can be decorated:

* Any managed bean that are not decorator or interceptor
* EJB session beans
* Any built-in bean provided by the CDI container except the @Default BeanManager type.
* The beans returned by a producer method or producer field **are not decorated.**

## How to define a decorator bean

* A decorator is a managed bean by its own. With dependent scope;
* The decorated types are defined as the set of interfaces implemented by the decorated bean (The decorator class and its superclasses are not considered as a decorated type. )
* A decorator class can be abstract and in this case it may not implement all methods defined by the decorated type.
* A decorator class must implement at least one decorated type (must declare at least one interface)
* The class must be annotated with the stereotype annotation: @javax.decorator.Decorator
* The class must have one and only one delegate injection point:
  + The decorator has access to the the instance that is going to be decorated througt the declaration of an delegate Injection Point. The injection point must have a type and qualifiers compatible with the delegated bean.
    - An delegate Injection point is declared with:
      * Annotation: @Inject
      * Annotation: @javax.decorator.Delegate
      * An optional list of qualifiers to specify the target delegated type
      * A type. This type must implement all decorated types declared by the Decorator class
      * It can be defined on a:
        + field,
        + constructor
        + initializer method.
* A decorator will only be applyed by the CDI Container if it is enabled. To enable a container it is possible to do in two ways:
  + To enable the decorator only to the beans defined on the same bean archive. The decorator class path should be specified on the beans.xml file with the decorators/class tags
  + To enable a decorator to be applyed for all application beans it the decorator bean class should be annotated with the javax.annotation.Priority.
* If more that one decorator is specified to a type the CDI will applyed them in the order given by the Priority annotation value (the lower are executed first) or by the order specified by the tags decorators/class on the beans.xml file.

* A decorator is called after the interceptors.
* A decorator can call any number of methods from the decorator type. An interceptor cannot.

# Interceptors

## What is

The invocation of managed CDI beans methods can be intercepted by the CDI container with the goal to give the invocation logic to a second method (defined on the interceptor). This second method can execute operations before and after the original intercepted method execution.

An interceptor can be applied on three types of interceptor points:

* Managed beans method invocations
* Container lifecycle callback events
* EJB timeout methods (not cover here)

## How to define

### Define an interceptor binding annotation

Like a qualifier annotation, each interceptor needs a specific annotation to connect the interceptor implementation and the place where the interceptor should be applied.

The annotation should have the following caracteristics:

* Retention: Runtime
* Target: At least Type, and in addition it may have Method
* Be annotated with @javax.interceptors.InterceptorBinding
* May have any number of member properties. The value of those properties will be taken by the CDI container to find a match between the interceptor implementation and interceptor declaration. If an annotation member is not to be taken by the matching algorithm it must be annotated in @javax.enterprise.utils.NonBinding.

### Define an interceptor class

The interceptor class is called by the CDI Engine when an intercepted method is invoked.

The interceptor class is defined as :

* Dependent managed bean
* It must be annotated with:
  + @javax.interceptor.Interceptor
  + One or more interceptor binding annotation, for which it will be responsible to implement the interception logic.
* One method
  + Returning an Object type
  + At least one parameter of type javax.interceptor.InvocationContext
  + Annotated with javax.interceptor.AroundInvoke for intercepting business methods or by a CDI event annotation for intercepting container lidecycle callbacks.
  + It must invoke the method javax.interceptor.IncocationContext.proceed() an return it return value (of object type)
  + The javax.interceptor.InvocationContext parameter allows to :
    - Get the parameters of the intercepted invocation, and even calls the interceptor class to change the values of the parameters.
    - Get a reference to the instance that has the intercepted method
    - Know the properties of the invocation method

By default the interceptors classes are not recognized by the CDI container. They must be enabled. Like the decorators, there are two ways to enable the interceptor classes:

* Placing the class on the beans.xml files using the tags: interceptors/class. The interceptors specified on a beans.xml file are only applied to the managed beans defined by the same bean archive. The specification order on the bean.xml is used by the container to define the order on which the interceptors are applied to the same method.
* Annotating the class with javax.annotation.Priority and specified a number. The interceptor class annotated with Priority can be applied to any application managed bean. The number on the annotation gives the interceptor order: a lower valuer is executed first than an higher value. The priority interceptor are executed before the bean.xml interceptors.

### Apply the interceptor on a regular managed bean

To apply an interceptor on a business managed bean is only necessary to annotate those class with the specific interceptor binding annotation. The the interceptor binding annotation is placed:

* On the class, then all methods will be intercepted
* On a specific method, then only this method will be intercepted.

A producer method can be also intercepted as normal business method. The interceptor annotation of a producer method are not applied to the produced bean.

A interceptor binding annotation can be places on stereotype annotation. However if a class has a stereotype annotation with a interceptor annotation and also a interceptor annotation then the stereotype interceptor annotation will not be taken into consideration.

The class with the interceptor binding annotation must be proxy capable.

# Events

## What is

The CDI specification offers a mechanism for applications to use a publish-subscribe communication pattern between the applications CDI beans: the CDI events. Inclusive, the CDI events are used by the CDI container to notify the application about its own internal workings, specialy the CDI container emits events through out its lifecycles states.

A CDI bean can emit or receive an event.

## How to use events

### Define a event type and event qualifiers

The event type defines the data that is going to be send by the emitter bean to the other observers beans. An event type is a concrete java class (cannot be a CDI instance) or any of its interfaces or superclasses. The event object must be a concrete class.

The event qualifier is like a qualifier annotation:

* Annotated with javax.inject.Qualifier
* Retention: Runtime
* Target: Method, Field,Parameter, Type. If an event type has always the qualifier Any even if does not have an explicit qualifier.
* May have members. This members are used by the matching algorithm to find out the observers for the event. If a member value should not be taken into consideration for the matching algorithm, then it should be annotated with javax.enterprise.util.NonBinding

### Fire an event

For a Bean to fire an event it must have access to CDI javax.enterprise.event.Event. through an injection point:

* The event object is a generic type that must be parameterized with the type of the event that is going to be fired.
* The Event inject point can also be qualified with a set of qualifiers annotations. This annotations specificy the qualifiers for the event that is going to be fired.

When the emitter beans wants to fire an event inside it business methods it just needs to execute the method:

* Event.fire(event objects)

Programmatically the emitter bean can redefine the injection point event qualifiers by specifing the extra annotations for the fired event using the Event.select method. This allows for an emitter bean to dynamically send different events, although with the same event type.

The action of firing an event is synchronous. When an emitter bean fires an event, it waits for the execution of all observer methods that might be on the application. Only then the firing managed bean resumes its business logic.

### Observe an event

An observer event receives the event on a method: the observer method. This method must have one parameter with the following characteristics:

* Defined on a managed bean or session bean
* Non-abstract, static or non-static, default-access, public, protected or private method.
* Annotated with: @javax.enterprise.event.Observes
* A optional set of event qualifiers annotations. By default an observer parameter has the @Any qualifier declared, if no user event qualifier is specified. If more than one event qualifier are present then the observer method will only be called when the fired event has at least all qualifiers defined on the observes parameter.
* A type compatible with the event type. The parameter type must be the same type as the event fired or one of the interface or superclass from the fired event typed.

An observer method can have other parameters. Those extra parameters are injection points. The observer method cannot by annotated with @inject.

An object method can declare a parameter of type javax.interface.spi.EventMetadata. The value of this parameter is injected by the CDI container and allows to found out information about the fired event. It has the methods:

* getQualifiers() : returns the set of qualifiers used to fire the event
* GetType() : returns the type of the event fired
* getInjectionPoint() : returns information about the injection point where the event was fired.

A managed bean can have any number of observer methods with the same event type and event qualifiers.

An instance from the managed bean with the observer method is created when a matching event is fired, if the instance is not already present on the bean scope. If this behaviour is not the wanted one, then the annotation @Observes has the member receive (receive=IF\_EXISTS) that allows to tell to the CDI container for not creating a bean instance when an event is fired.

An Interceptor or Decorated cannot have observer methods.

The order of execution of obsersables methods is not defined.

# Extensions

## Container API

An application can have access to a CDI bean provided by the CDI container that offers a set of methods to interact with the CDI container. Exists two ways to have access to the bean of type javax.enterprise.inject.spi.BeanManager :

* Through an injection point: @Inject BeanManager beanManager
* Programatically calling the static method: javax.enterprise.inject.spi.CDI.current().getBeanManager().

The BeanManager offers the following functionality:

* Retrieve the CDI information about one bean, like its name, its scope, its injection points and so on.
* Retrieve the CDI information about one Decorator , Interceptor, ObserverMethod .
* Creating and registering dynamically a Bean or Producer definition.
* Obtaining a reference to contextual reference for a bean.

## Extensions

The CDI Container allows to third party libraries to extend the container functionality. Those third party libraries will observe a specific set of events fired by the container. The events are fired at container startup and shutdown. Each fired event as a well established place in a predefined sequence of events and have a specific event type (payload). The events are fired and observed the same way as the user events.

For an extension third party library to observe the lifecycle events it must be a java service provider for the service: javax.enterprise.inject.spi.Extension. To do that, it must have :

* A directory META-INF/services
* Inside the directory a file with name: javax.enterprise.inject.spi.Extension
* The file must contain one line with the full path for a class that implements the interface, jaxax.enterprise.inject.spi.Extension. This class can have any number of observer methods for the lifecycle events.

An extension class is instanciated at the container startup and destroed when the container shutdowns. Througth the container lifecycle any CDI Bean can have an injected point for an extension class (the bean has an Application scope)