# Java Enterprise Edition

IQSOFT-John Bryce Training Center

Dr. Attila Vincze

May 13 - May 17, 2019





## Outline I

# CDI 1.1 I

Introduction

- Inversion of control (IoC), meaning that the container would take control of your business code and provide technical services (such as transaction or security management).
- Taking control meant managing the life cycle of the components, bringing dependency injection and configuration to your components.
- CDI is built on the concept of "loose coupling, strong typing," meaning that beans are loosely coupled but in a strongly typed way

Understanding Beans

- Java SE has JavaBeans, Java EE has Enterprise JavaBeans.
- Java EE has other sorts of components such as Servlets, SOAP web services, RESTful web services, entities and of course Managed Beans.
- POJOs are just Java classes that run inside the Java Virtual Machine (JVM).
- JavaBeans are just POJOs that follow certain patterns (e.g., a naming convention for accessors/mutators (getters/setters) for a property, a default constructor ) and are executed inside the JVM.
- Managed Beans are container-managed objects that support only a small set of basic services: resource injection, life-cycle management, and interception.





# CDI 1.1 II

Dependency Injection

- Dependency Injection (DI) is a design pattern that decouples dependent components.
- It is part of inversion of control, where the concern being inverted is the process of obtaining the needed dependency.
- Instead of an object looking up other objects, the container injects those dependent objects for you.
- This is the so-called Hollywood Principle, "Don’t call us? "(lookup objects), "we’ll call you" (inject objects).
- Java EE 5 introduced a new set of annotations (@Resource, @PersistenceContext, @PersistenceUnit, @EJB, and @WebServiceRef).

Life-Cycle Management

- The life cycle of a POJO is pretty simple: as a Java developer you create an instance of a class using the new keyword and wait for the Garbage Collector to get rid of it and free some memory.
- If you want to run a CDI Bean inside a container, you are not allowed to use the new keyword.
- You need to inject the bean and the container does the rest, meaning, the container is the one responsible for managing the life cycle of the bean: it creates the instance; it gets rid of it.

# CDI 1.1 III

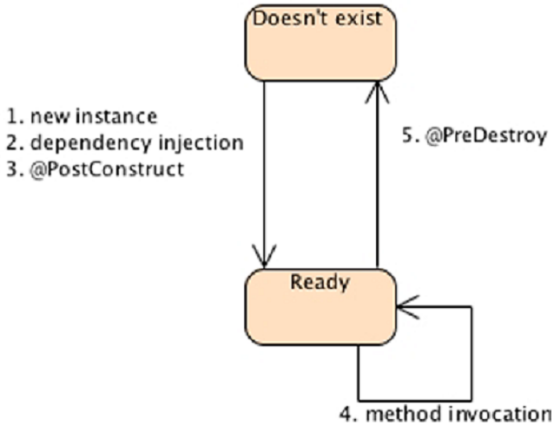


Figure: Managed Bean life cycle





# CDI 1.1 IV

Scopes and Context

- CDI Beans may be stateful and are contextual, meaning that they live in a well-defined scope (CDI comes with predefined scopes: request, session, application, and conversation scopes).
- The container manages all beans inside the scope automatically for you and, at the end of the session, automatically destroys them.
- Different clients of a stateful bean see the bean in different states.

# CDI 1.1 V

Interception

- Interceptors are used to interpose on business method invocations.
- In this aspect, it is similar to aspect-oriented programming (AOP). AOP is a programming paradigm that separates cross-cutting concerns (concerns that cut across the application) from your business code.
- Managed Beans support AOP-like functionality by providing the ability to intercept method invocation through interceptors.
- When you develop a session bean, you just concentrate on your business code.
- Behind the scenes, when a client invokes a method on your EJB, the container intercepts the invocation and applies different services (life-cycle management, transaction, security, etc.).
- With interceptors, you add your own cross-cutting mechanisms and apply them transparently to your business code.





# CDI 1.1 VI

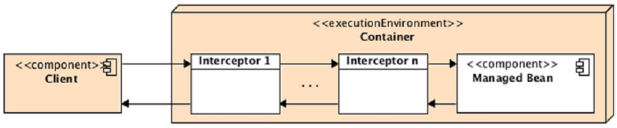


Figure: A container intercepting a call and invoking interceptors

# CDI 1.1 VII

Loose Coupling and Strong Typing

- With injection a bean is not aware of the concrete implementation of any bean it interacts with.
- Beans can use event notifications to decouple event producers from event consumers or decorators to decouple business concerns.
-





# CDI 1.1 VIII

Deployment Descriptor

- Nearly every Java EE specification has an optional XML deployment descriptor.
- With CDI, the deployment descriptor is called beans.xml and is mandatory.
- At deployment time, CDI checks all of your application's jar and war files and each time it finds a beans.xml deployment descriptor it manages all the POJOs, which then become CDI Beans.
- If your web application contains several jar files and you want to have CDI enabled across the entire application, each jar will need its own beans.xml to trigger CDI and bean discovery for each jar.

# CDI 1.1 IX

A CDI Bean can be any kind of class that contains business logic. It may be called directly from Java code via injection, or it may be invoked via EL from a JSF page.

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      @Inject
      private EntityManager em;
      private Date instanciationDate;
      @PostConstruct
      private void initDate() {
            instanciationDate = new Date();
      }
      @Transactional
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            book.setInstanciationDate(instanciationDate);
            em.persist(book);
            return book;
      }
}
```

A BookService Bean Using Injection Life-Cycle Management and Interception

Anatomy of a CDI Bean

- It is not a non-static inner class





# CDI 1.1 X

- It is a concrete class, or is annotated @Decorator
- It has a default constructor with no parameters, or it declares a constructor annotated @Inject.

```
public class BookService {
      private NumberGenerator numberGenerator;
      public BookService() {
            this.numberGenerator = new IsbnGenerator();
      }
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}
```

A BookService POJO Creating Dependencies Using the New Keyword

# CDI 1.1 XI

```
public class BookService {
      private NumberGenerator numberGenerator;
      public BookService(NumberGenerator numberGenerator) {
            this.numberGenerator = numberGenerator;
      }
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}

BookService bookService = new BookService(new IsbnGenerator());
BookService bookService = new BookService(new IssnGenerator());
```

A BookService POJO Choosing Dependencies Using the Constructor

**This illustrates what inversion of control is: the control of creating the dependency between BookService and NumberGenerator is inverted because it's given to an external class, not the class itself. Since you end up connecting the dependencies yourself, this technique is referred to as construction by hand. we used the constructor to choose implementation (constructor injection), but another common way is to use setters (setter injection).**





# CDI 1.1 XII

@Inject

- As Java EE is a managed environment you don't need to construct dependencies by hand but can leave the container to inject a reference for you.
- In a nutshell, CDI dependency injection is the ability to inject beans into others in a typesafe way, which means no XML but annotations.
- Injection already existed in Java EE 5 with the @Resource, @PersistentUnit or @EJB annotations, for example. But it was limited to certain resources (datasource, EJB . . .) and into certain components (Servlets, EJBs, JSF backing bean . . .).
- With CDI you can inject nearly anything anywhere thanks to the @Inject annotation.

# CDI 1.1 XIII

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}

public class IsbnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "13-84356-" + Math.abs(new Random().nextInt());
      }
}
```

BookService Using @Inject to Get a Reference of NumberGenerator





# CDI 1.1 XIV

Injection can occur via three different mechanisms: property, setter, or constructor. Notice that it isn't necessary to create a getter and a setter method on an attribute to use injection. CDI can access an injected field directly (even if it's private), which sometimes helps eliminate some wasteful code.

```
@Inject
private NumberGenerator numberGenerator;

@Inject
public BookService (NumberGenerator numberGenerator) {
      this.numberGenerator = numberGenerator;
}

@Inject
public void setNumberGenerator(NumberGenerator numberGenerator) {
      this.numberGenerator = numberGenerator;
}
```

The container is the one doing injection, not you (you can't invoke a constructor in a managed environment); therefore, there is only one bean constructor allowed so that the container can do its work and inject the right references.

# CDI 1.1 XV

Default Injection

- Assume that NumberGenerator only has one implementation (IsbnGenerator). CDI will then be able to inject it simply by using @Inject on its own.

Whenever a bean or injection point does not explicitly declare a qualifier, the container assumes the qualifier @javax.enterprise.inject.Default.

```
@Inject
private NumberGenerator numberGenerator;

@Inject @Default
private NumberGenerator numberGenerator;
```

@Default is a built-in qualifier that informs CDI to inject the default bean implementation. If you define a bean with no qualifier, the bean automatically has the qualifier @Default

```
@Default
public class IsbnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "13-84356-" + Math.abs(new Random().nextInt());
      }
}
```

The IsbnGenerator Bean with the @Default Qualifier





# CDI 1.1 XVI

If you only have one implementation of a bean to inject, the default behavior applies and a straightforward @Inject will inject the implementation.

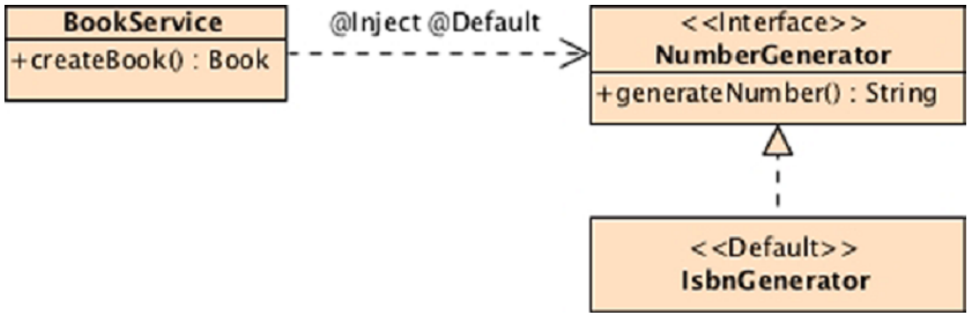


Figure: Class diagram with @Default injection

# CDI 1.1 XVII

Qualifiers

- At system initialization time, the container must validate that exactly one bean satisfying each injection point exists. Meaning that if no implementation of NumberGenerator is available, the container would inform you of an unsatisfied dependency and will not deploy the application.
- If there is only one implementation, injection will work using the @Default qualifier.
- If more than one default implementation were available, the container would inform you of an ambiguous dependency and will not deploy the application. That's because the typesafe resolution algorithm fails when the container is unable to identify exactly one bean to inject.





# CDI 1.1 XVIII

**CDI uses qualifiers, which basically are Java annotations that bring typesafe injection and disambiguate a type without having to fall back on String-based names.**

# CDI 1.1 XIX

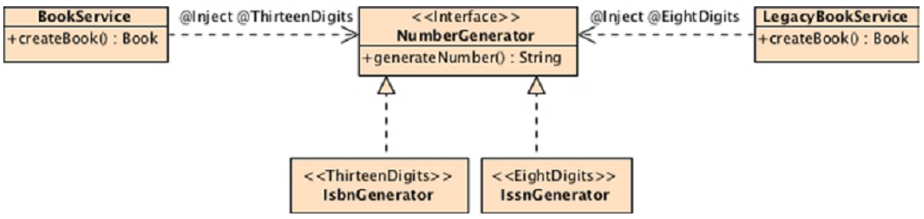


Figure: Services using qualifiers for non-ambiguous injection

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

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





# CDI 1.1 XX

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "13-84356-" + Math.abs(new Random().nextInt());
      }
}

@EightDigits
public class IssnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "8-" + Math.abs(new Random().nextInt());
      }
}
```

# CDI 1.1 XXI

```
public class BookService {
      @Inject @ThirteenDigits
      private NumberGenerator numberGenerator;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}

public class LegacyBookService {
      @Inject @EightDigits
      private NumberGenerator numberGenerator;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}
```





# CDI 1.1 XXII

## Qualifiers with Members

```
@Qualifier
@Retention(RUNTIME) @Target({
      FIELD, TYPE, METHOD}) public @interface NumberOfDigits {
      Digits value();
      boolean odd();
}
public enum Digits {
      TWO,
      EIGHT,
      TEN,
      THIRTEEN
}
```

```
@Inject @NumberOfDigits(value = Digits.THIRTEEN, odd = false)
private NumberGenerator numberGenerator;

@NumberOfDigits(value = Digits.THIRTEEN, odd = false)
public class IsbnEvenGenerator implements NumberGenerator {
}
```

## Multiple Qualifiers

# CDI 1.1 XXIII

```
@ThirteenDigits @Even
public class IsbnEvenGenerator implements NumberGenerator {
}

@Inject @ThirteenDigits @Even
private NumberGenerator numberGenerator;
```

Alternatives

- Qualifiers let you choose between multiple implementations of an interface at development time. But sometimes you want to inject an implementation depending on a particular deployment scenario. For example, you may want to use a mock number generator in a testing environment.
- Alternatives are beans annotated with the special qualifier javax.enterprise.inject.Alternative. By default alternatives are disabled and need to be enabled in the beans.xml descriptor to make them available for instantiation and injection.





# CDI 1.1 XXIV

```
@Alternative
public class MockGenerator implements NumberGenerator {
      public String generateNumber() {
            return "MOCK";
      }
}

@Alternative @Default
public class MockGenerator implements NumberGenerator {
}

@Alternative @ThirteenDigits
public class MockGenerator implements NumberGenerator {
}
```

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee ÂŹ
 http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
    <alternatives>
        <class>org.agoncal.book.javaee7.chapter02. MockGenerator</class>
    </alternatives>
</beans>
```

# CDI 1.1 XXV

```
@Inject
private NumberGenerator numberGenerator;
```

You can have several beans.xml files declaring several alternatives depending on your environment (development, production, test).





# CDI 1.1 XXVI

Producers

- I've shown you how to inject CDI Beans into other CDI Beans. But you can also inject primitives (e.g., int, long, float), array types and any POJO that is not CDI enabled, thanks to producers. By CDI enabled I mean any class packaged into an archive containing a beans.xml file.
- By default, you cannot inject classes such as a java.util.Date or java.lang.String. That's because all these classes are packaged in the rt.jar file (the Java runtime environment classes) and this archive does not contain a beans.xml deployment descriptor.
- The only way to be able to inject POJOs is to use producer fields or producer methods.

```
public class NumberProducer {
      @Produces @ThirteenDigits
      private String prefix13digits = "13-";

      @Produces @ThirteenDigits
      private int editorNumber = 84356;

      @Produces @Random
      public double random() {
            return Math.abs(new Random().nextInt());
      }
}
```

# CDI 1.1 XXVII

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {

      @Inject @ThirteenDigits private String prefix;

      @Inject @ThirteenDigits private int editorNumber;

      @Inject @Random
      private double postfix;
      public String generateNumber() {
            return prefix + editorNumber + postfix;
      }
}
```





# CDI 1.1 XXVIII

InjectionPoint API

- There are certain cases where objects need to know something about the injection point into which they are injected.
- How would you produce a Logger that needs to know the class name of the injection point?
- CDI has an InjectionPoint API that provides access to metadata about an injection point.

```
Logger log = Logger.getLogger(BookService.class.getName());
```

```
Logger log = Logger.getLogger(BookService.class.getName());

public class LoginProducer {

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

To use the produced logger in any bean you just inject it and use it. The logger's category class name will then be automatically set

```
@Inject Logger log;
```

# CDI 1.1 XXIX

Disposers

- Some producer methods return objects that require explicit destruction such as a Java Database Connectivity (JDBC) connection, JMS session, or entity manager.
- For creation, CDI uses producers, and for destruction, disposers.
- A disposer method allows the application to perform the customized cleanup of an object returned by a producer method.

```
public class JDBCConnectionProducer {
      @Produces
      private Connection createConnection() {
            Connection conn = null;
            try {
                  Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
                  conn = DriverManager.getConnection("jdbc:derby:memory:chapter02DB", "APP", "APP");
            } catch (InstantiationException | IllegalAccessException | ClassNotFoundException) {
                  e.printStackTrace();
            }
            return conn;
      }
      private void closeConnection(@Disposes Connection conn) throws SQLException {
            conn.close();
      }
}
```





# CDI 1.1 XXX

- Destruction can be performed by a matching disposer method, defined by the same class as the producer method.
- Each disposer method, annotated with @Disposes, must have exactly one disposed parameter of the same type (here java.sql.Connection) and qualifiers (@Default) as the corresponding producer method return type (annotated @Produces).
- The disposer method (closeConnection() ) is called automatically when the client context ends and the parameter receives the object produced by the producer method.

```
@ApplicationScoped
public class DerbyPingService {
      @Inject
      private Connection conn;
      public void ping() throws SQLException {
            conn.createStatement().executeQuery("SELECT 1 FROM SYSIBM.SYSDUMMY1");
      }
}
```

# CDI 1.1 XXXI

Scopes

- Every object managed by CDI has a well-defined scope and life cycle that is bound to a specific context.
- In Java, the scope of a POJO is pretty simple: you create an instance of a class using the new keyword and you rely on the garbage collection to get rid of it and free some memory.
- With CDI, a bean is bound to a context and it remains in that context until the bean is destroyed by the container. There is no way to manually remove a bean from a context.

CDI defines the following built-in scopes and even gives you extension points so you can create your own

- Application scope (@ApplicationScoped)
- Session scope (@SessionScoped)
- Request scope (@RequestScoped)
- Conversation scope (@ConversationScoped)
- Dependent pseudo-scope (@Dependent)





# CDI 1.1 XXXII

Conversation

- The conversation scope is slightly different than the application, session, or request scope. It holds state associated with a user, spans multiple requests, and is demarcated programmatically by the application.

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable {
      private Login login;
      private Account account;

      @Inject
      private CustomerService customerService;

      @Inject
      private Conversation conversation;

      public void saveLogin() {
            conversation.begin();
            login = new Login();
            // Sets login properties
      }
      public void saveAccount() {
            account = new Account();
            // Sets account properties
      }
```

# CDI 1.1 XXXIII

```
        public void createCustomer() {
                Customer customer = new Customer();
                customer.setLogin(login);
                customer.setAccount(account);
                customerService.createCustomer(customer);
                conversation.end();
        }
}
```





# CDI 1.1 XXXIV

## Beans in Expression Language

```
@Named
public class BookService {
        private String title, description;
        private Float price;
        private Book book;
        @Inject @ThirteenDigits
        private NumberGenerator numberGenerator;
        public String createBook() {
                book = new Book(title, price, description);
                book.setIsbn(numberGenerator.generateNumber());
                return "customer.xhtml";
        }
}
```

```
<h:commandButton value="Send email" action="#{
        bookService.createBook}"/>
```

```
@Named("myService")
public class BookService {
}
```

```
<h:commandButton value="Send email" action="#{
        myService.createBook}"/>
```

# CDI 1.1 XXXV

Interceptors

Constructor-level interceptors Interceptor associated with a constructor of the target class (@AroundConstruct)

Method-level interceptors Interceptor associated with a specific business method (@AroundInvoke)

Timeout method interceptors Interceptor that interposes on timeout methods with (@AroundTimeout)

Life-cycle callback interceptors Interceptor that interposes on the target instance life-cycle event callbacks (@PostConstruct and @PreDestroy)





# CDI 1.1 XXXVI

```
@Transactional
public class CustomerService {
      @Inject
      private EntityManager em;
      @Inject
      private Logger logger;
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
      @AroundInvoke
      private Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

A CustomerService Using Around-Invoke Interceptor

# CDI 1.1 XXXVII

```
public class LoggingInterceptor {
      @Inject
      private Logger logger;
      @AroundConstruct
      private void init(InvocationContext ic) throws Exception {
            logger.fine("Entering constructor");
            try {
                  ic.proceed();
            } finally {
                  logger.fine("Exiting constructor");
            }
      }
      @AroundInvoke
      public Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

An Interceptor Class with Around-Invoke and Around-Construct





# CDI 1.1 XXXVIII

```
@Transactional
public class CustomerService {
      @Inject
      private EntityManager em;
      @Interceptors(LoggingInterceptor.class)
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
}
```

CustomerService Uses an Interceptor on One Method

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
      public void createCustomer(Customer customer) {
            ...}
      public Customer findCustomerById(Long id) {
            ...}
}
```

# CDI 1.1 XXXIX

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
      public void createCustomer(Customer customer) {
      }
      public Customer findCustomerById(Long id) {
      }
      @ExcludeClassInterceptors
      public Customer updateCustomer(Customer customer) {
      }
}
```

Life-Cycle Interceptor





# CDI 1.1 XL

```
public class ProfileInterceptor {
      @Inject
      private Logger logger;
      @PostConstruct
      public void logMethod(InvocationContext ic) throws Exception {
            logger.fine(ic.getTarget().toString());
            try {
                  ic.proceed();
            } finally {
                  logger.fine(ic.getTarget().toString());
            }
      }
      @AroundInvoke
      public Object profile(InvocationContext ic) throws Exception {
            long initTime = System.currentTimeMillis();
            try {
                  return ic.proceed();
            } finally {
                  long diffTime = System.currentTimeMillis() - initTime;
                  logger.fine(ic.getMethod() + " took " + diffTime + " millis");
            }
      }
}
```

An Interceptor with Both Life-Cycle and Around-Invoke

# CDI 1.1 XLI

```
@Transactional
@Interceptors(ProfileInterceptor.class)
public class CustomerService {
      @Inject
      private EntityManager em;
      @PostConstruct
      public void init() {
            // ...
      }
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
}
```

CustomerService Using an Interceptor and a Callback Annotation

Chaining and Excluding Interceptors





# CDI 1.1 XLII

```
@Stateless
@Interceptors({
      I1.class, I2.class})
public class CustomerService {
      public void createCustomer(Customer customer) {
            ...}
      @Interceptors({
            I3.class, I4.class})
      public Customer findCustomerById(Long id) {
            ...}
      public void removeCustomer(Customer customer) {
            ...}
      @ExcludeClassInterceptors
      public Customer updateCustomer(Customer customer) {
            ...}
}
```

CustomerService Chaining Serveral Interceptors

# CDI 1.1 XLIII

## Interceptor Binding

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

```
@Interceptor
@Loggable
public class LoggingInterceptor {
      @Inject
      private Logger logger;
      @AroundInvoke
      public Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

Loggable Interceptor





# CDI 1.1 XLIV

```
@Transactional
@Loggable
public class CustomerService {
      @Inject
      private EntityManager em;
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
}
```

CustomerService using the Interceptor Binding

```
@Transactional
public class CustomerService {
      @Loggable
      public void createCustomer(Customer customer) {
            ...}
      public Customer findCustomerById(Long id) {
            ...}
}
```

# CDI 1.1 XLV

## Prioritizing Interceptors Binding

- Interceptor binding brings you a level of indirection, but you lose the possibility to order the interceptors (@Interceptors(I1.class, I2.class) ).

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
      @Inject
      private Logger logger;
      @AroundInvoke
      public Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

```
@Interceptor
@Loggable
@Priority(Interceptor.Priority.LIBRARY_BEFORE + 10)
public class LoggingInterceptor {
}
```





# CDI 1.1 XLVI

## Decorator

```
@Decorator
public class FromEightToThirteenDigitsDecorator implements NumberGenerator {
      @Inject @Delegate
      private NumberGenerator numberGenerator;
      public String generateNumber() {
            String issn = numberGenerator.generateNumber(); String isbn = "13-84356" + issn.substring(1);
      }
}
```

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xs
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
version="1.1" bean-discovery-mode="all">

<decorators>
<class>org.agoncal.book.javaee7.chapter02.FromEightToThirteenDigitsDecorator</class>

</decorators>

</beans>
```

# CDI 1.1 XLVII

Events

- One bean can define an event, another bean can fire the event, and yet another bean can handle the event.

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      @Inject
      private Event<Book> bookAddedEvent;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            bookAddedEvent.fire(book);
            return book;
      }
}
```

```
public class InventoryService {
      @Inject
      private Logger logger;
      List<Book> inventory = new ArrayList<>();
      public void addBook(@Observes Book book) {
            logger.info("Adding book " + book.getTitle() + " to inventory"); inventory.add(book);
      }
}
```





# CDI 1.1 XLVIII

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      @Inject @Added
      private Event<Book> bookAddedEvent;
      @Inject @Removed
      private Event<Book> bookRemovedEvent;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber()); bookAddedEvent.fire(book);
            return book;
      }
      public void deleteBook(Book book) {
            bookRemovedEvent.fire(book);
      }
}
```

# CDI 1.1 XLIX

```
public class InventoryService {
      @Inject
      private Logger logger;
      List<Book> inventory = new ArrayList<>();
      public void addBook(@Observes @Added Book book) {
            logger.warning("Adding book " + book.getTitle() + " to inventory");
            inventory.add(book);
      }

      public void removeBook(@Observes @Removed Book book) {
            logger.warning("Removing book " + book.getTitle() + " to inventory");
            inventory.remove(book);
      }
}

void addBook(@Observes @Added @Price(greaterThan=100) Book book)
```