Context and Dependency Injection (CDI) Advanced

Sang Shin
JPassion.com
"Code with Passion!"

Topics

- Producer
- Events
- Alternatives
- Interceptor
- Decorator
- Stereotype

Producer

Producer Method (@Produces)

- Producer methods allow control over the production of the dependency objects
  - Runtime polymorphism – Producer can return different object depending on runtime condition (business logic, time of the day, etc)
  - Creation of Non-bean instance - Objects to be injected are not required to be instances of beans
  - Custom setup - Objects require some custom initialization that is not performed by the bean constructor
- Enables decoupling of a “producer” from the “consumer” (of dependency objects)
- Enables type-safe programming model for injecting Java EE resources

Runtime Polymorphism

```
// Inject Coder object – consumer of the Coder does not need to know how it gets created
@Inject
@Chosen    // This is qualifier, in other words, I want @Chosen + Coder type object to be injected
@RequestScoped
Coder coder;

// Producer method returns @Chosen + Coder type object but returns a different object
// depending on a runtime condition
@Produces
@Chosen
@RequestScoped
public Coder getCoder(@New TestCoderImpl tci, @New CoderImpl ci) {

   switch (coderType) {  // codeType value is set by business logic
   case TEST:
        return tci;

   case SHIFT:
        return ci;

   default:
      return null;
   }
}
```

Creation of Non-Bean Instance

```
// Inject FacesContext object
@Inject FacesContext myFacesContext;


// Produce FacesContext object that we don't control
@Produces
@RequestScope
public FacesContext  getFacesContext () {

    return FacesContext.getInstance();

  }
}
```

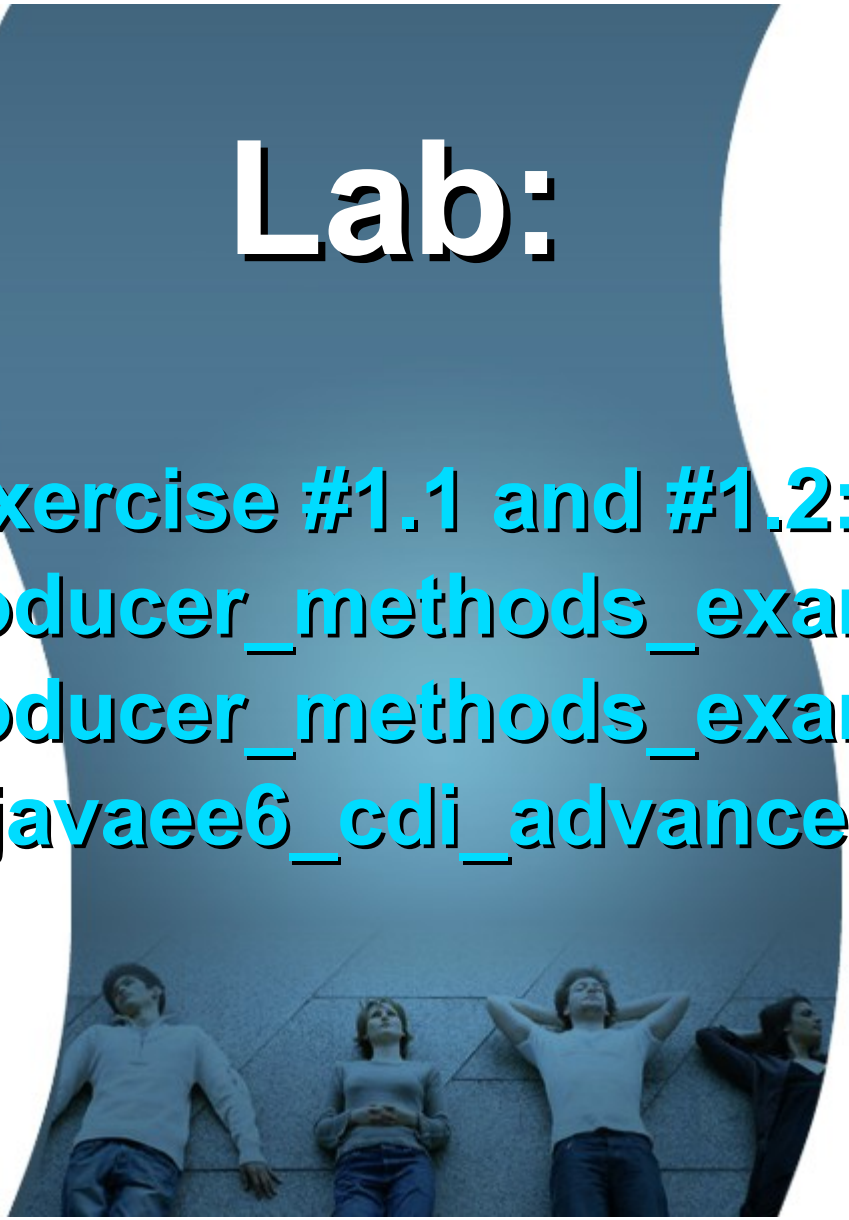
Disposer method with @Disposes

- Used to dispose objects that is produced and need to be explicitly destroyed
  - For example, a JDBC connection
- Disposer method takes to-be-disposed object with *@Disposes* annotation

```
@Produces @RequestScoped Connection connect(User user) {
   return createConnection(user.getId(), user.getPassword());
}

// The disposer method is called automatically when the context ends
// (in this case, at the end of the request), and this parameter receives
// the object produced by the producer method
void close(@Disposes Connection connection) {
   connection.close();
}
```

Lab:

Exercise #1.1 and #1.2:
cdi_producer_methods_example1
cdi_producer_methods_example2
4533_javaee6_cdi_advanced.zip



Type-safe Use of Java EE Resources (1)

- Use Producer field to expose Java EE resources in type-safe way

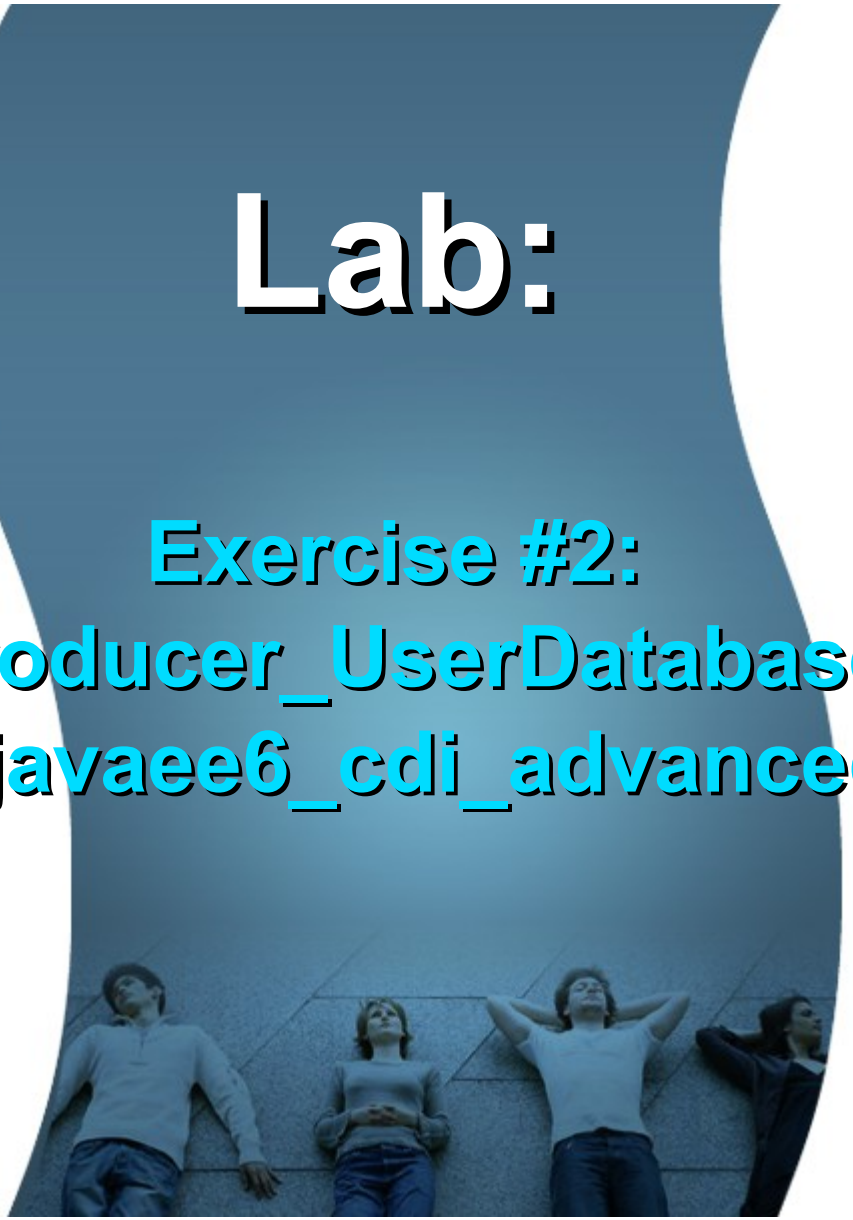
```
// Define a qualifier called UserDatabase
@Qualifier
@Retention(RUNTIME)
@Target({
    METHOD, FIELD, PARAMETER, TYPE
})
public @interface UserDatabase {
}

// Expose Java EE resource (EntityManager) via Producer
@Singleton
public class UserDatabaseEntityManager {
    @Produces
    @UserDatabase
    @PersistenceContext(unitName="producerfieldsPU")
    private EntityManager em;
}
```

Type-safe Use of Java EE Resources (2)

```
@ConversationScoped
@Stateful
public class RequestBean {

    // Inject Java EE resource in type-safe way, no more string
    @Inject @UserDatabase
    EntityManager em;

    public ToDo createToDo(String inputString) {
        ...

        try {
            ...
            em.persist(toDo);

            return toDo;
        } catch (Exception e) {
            throw new EJBException(e.getMessage());
        }
    }
```

Lab:

Exercise #2:
cdi_producer_UserDatabase_db
4533_javaee6_cdi_advanced.zip



Events

CDI Event Observer Pattern

- Completely decouple action (event producer) and reactions (event consumers)
  - No compile-time dependency between event producer and event consumer
- The event object acts as a payload, to propagate state from producer to consumer
- You can use Qualifiers that act as event selectors
  - Allows the consumer to narrow the set of events it observes through the use of qualifier
- Can be observed
  - Immediately (we will cover only this in this presentation)
  - After transaction completion
  - Asynchronously (through extension)

CDI Event Observer Pattern

- Define Event Class
- Event producer fires an event
- Event consumer observes event through *@Observes*

Define Event Class

- Event class can be any POJO class

```
public class LoggedInEvent {
    private String user;

    public LoggedInEvent(String user) {
        this.user = user;
    }
}
```

Event Producers

- Event producer fires an event through <Injected-Event-Object>.fire(<New-Event-Object>) method

```
public class Login {

    @Inject
    Event<LoggedInEvent> loggedInEvent;

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

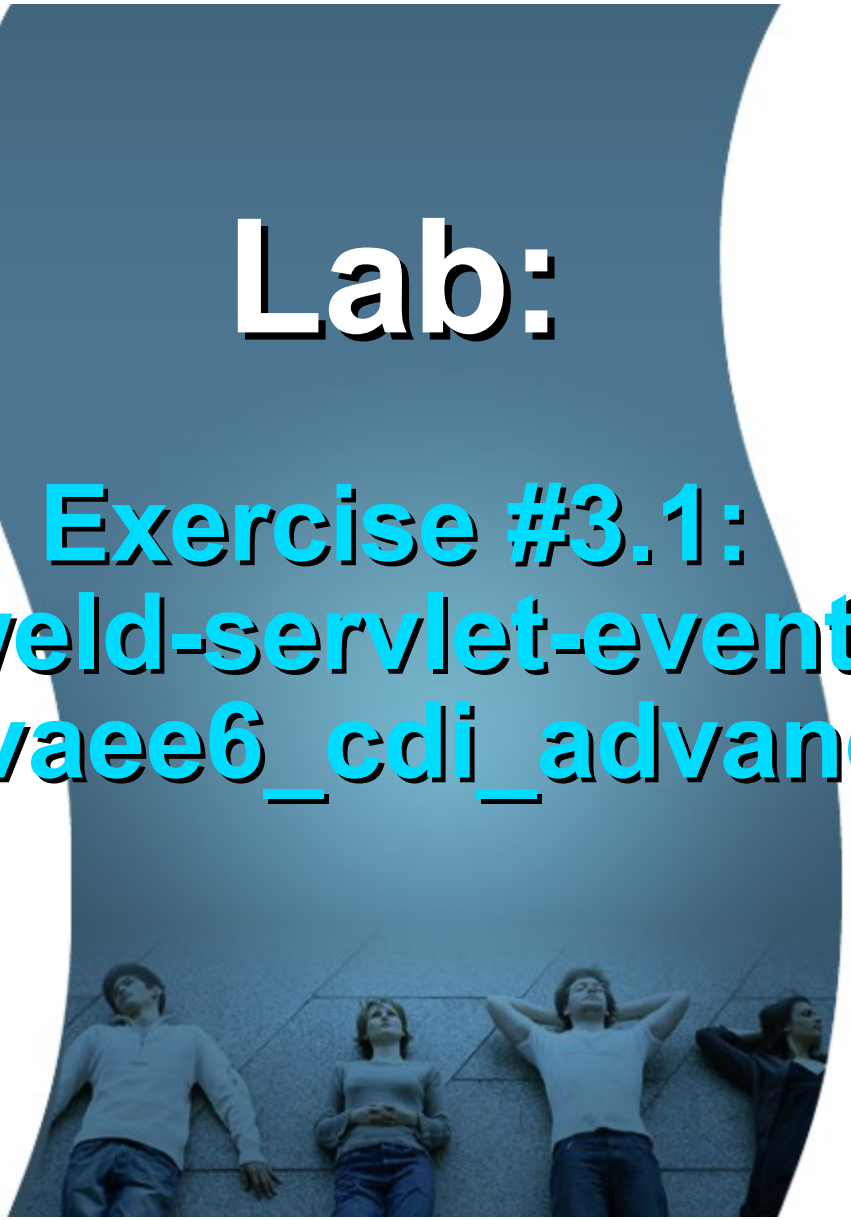
Event Consumer (Event Observer)

- The only thing event consumer has to do is to use *@Observes* <Event-class> annotation

```
// This methog gets invoked when LoggedInEvent is fired
public void afterLogin(@Observes LoggedInEvent event) {
    System.out.println("afterLogin() method is called, event = " + event);
}
```

Lab:

Exercise #3.1: weld-servlet-event 4533_javaee6_cdi_advanced.zip



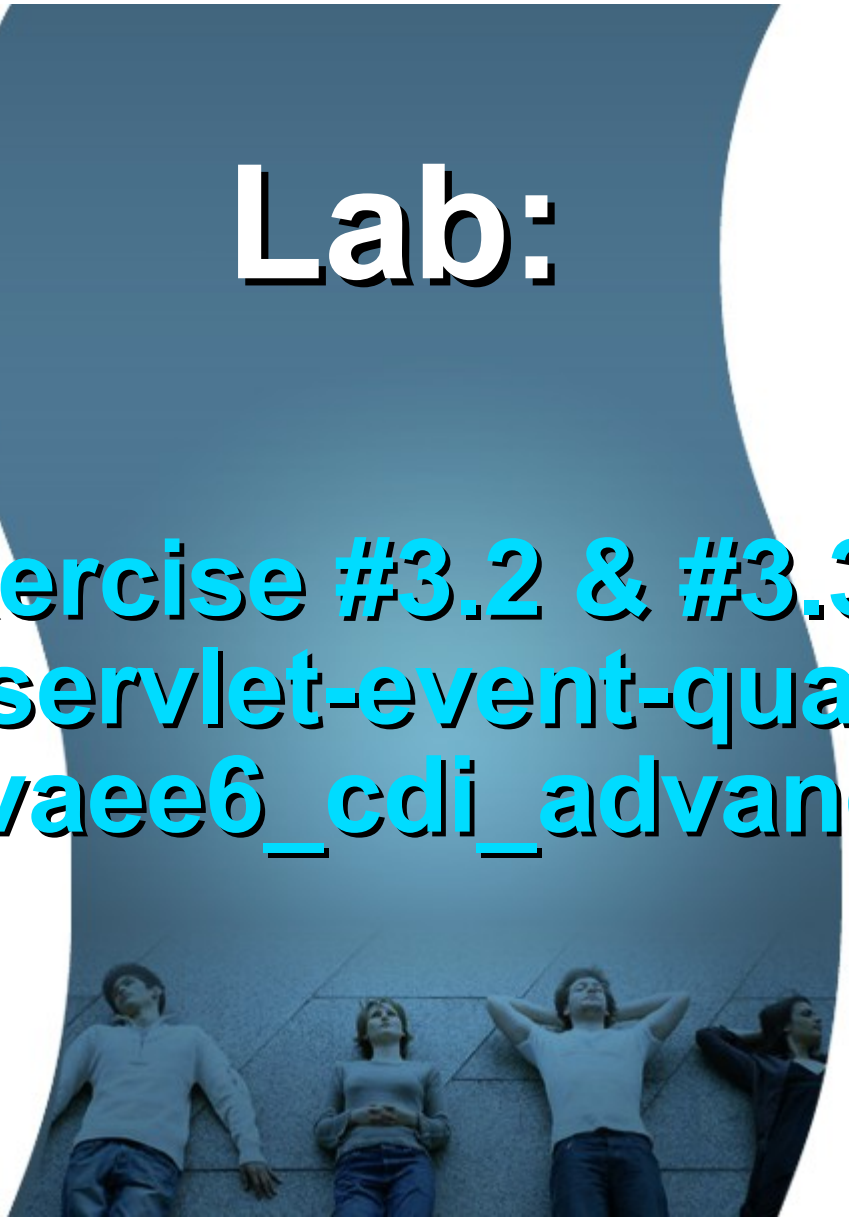
Event with Qualifier

- Event can be selectively fired and received through qualifier

```
public class Login {

    @Inject
    @Admin
    Event<LoggedInEvent> loggedInEvent;

    // Event producer
    public void login() {
        loggedInEvent.fire(
            new LoggedInEvent(credentials.getUsername()));
    }
}

// Event consumer
public void afterAdminLogin(@Observes @Admin LoggedInEvent event) {
        System.out.println("----afterAdminLogin() method is called, event = " + event);
}
```

Lab:

Exercise #3.2 & #3.3:
weld-servlet-event-qualifier
4533_javaee6_cdi_advanced.zip



Alternative

What is Alternative Bean?

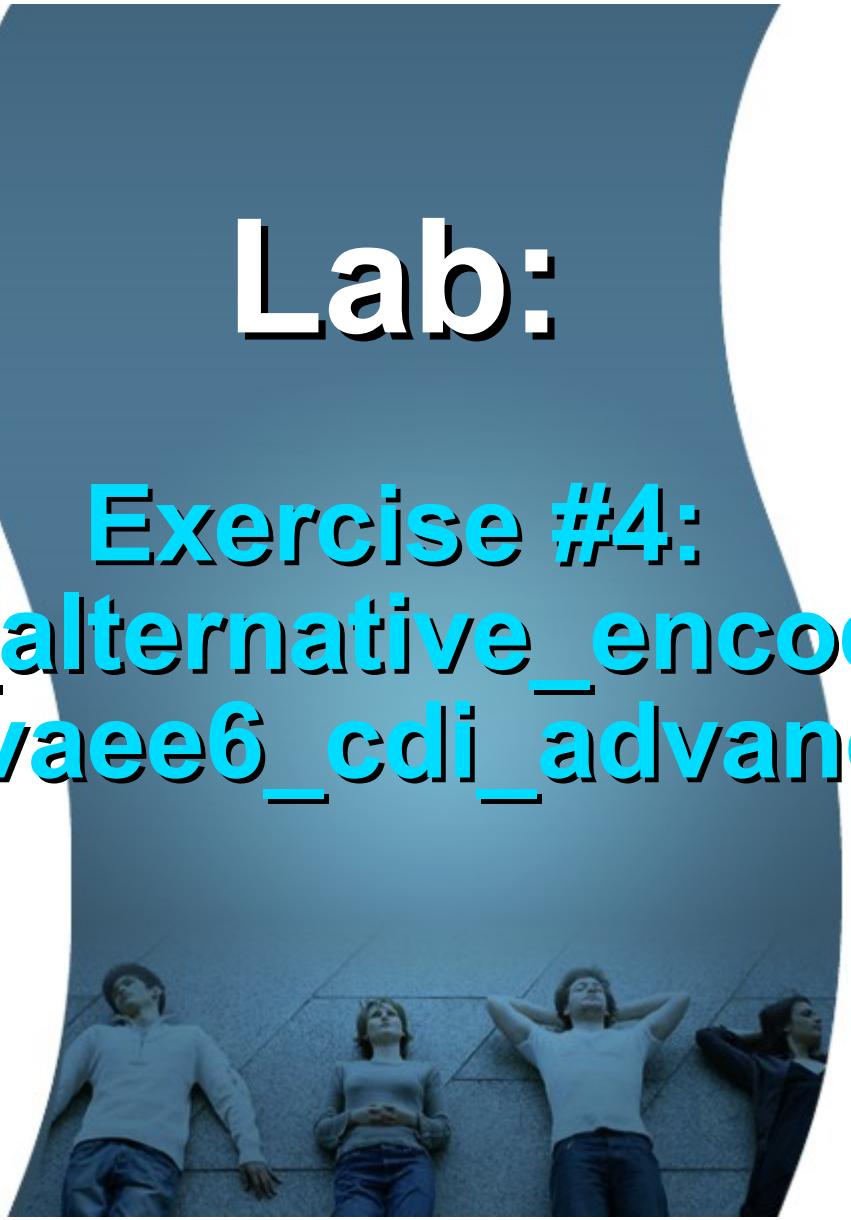
- Any bean with *@Alternative* is not considered for injection
  - Lets you package multiple beans that match injection type without ambiguity errors
  - In order to be considered for injection, it has to be activated in "*beans.xml*"
- Provide a replacement implementation during deployment
  - You can apply the *@Alternative* annotation to two or more beans, then, based on your deployment, specify the bean you want to use in the "*beans.xml*" configuration file
  - Useful for providing mock objects for testing – mock objects are annotated with *@Alternative*
    - In normal operation, beans with @Alternative will not be considered for injection
    - In testing operation, activate it via "beans.xml"

Alternative Bean

```
// Annotate alternative implementation with @Alternative annotation
@Alternative
public class TestCoderImpl implements Coder {

    public String codeString(
        String s,
        int tval) {
        return ("input string is " + s + ", shift value is " + tval);
    }
}

// Activate it in “beans.xml”
<beans>
    <alternatives>
        <class>encoder.TestCoderImpl</class>
    </alternatives>
</beans>
```

Lab:

Exercise #4:
cdi_alternative_encoder
4533_javaee6_cdi_advanced.zip



Interceptor

What is an Interceptor?

- Improves the Java EE 5 non-CDI interceptor scheme
- Interceptor wiring steps
  - #1: Define Interceptor binding
  - #2: Write interceptor implementation
  - #3: Apply interceptor to the target
- Interceptor has to be activated in "*beans.xml*"

Issues of non-CDI based Interceptor

- Interceptor implementation references are kind of hard-coded with the target
- The order of interceptor classes might not be guaranteed
- Looks verbose

```
// Example of non-CDI based interceptor scheme
@Interceptors(
    SecurityInterceptor.class,
    TransactionInterceptor.class,
    LoggingInterceptor.class
)
@Stateful
public class BusinessComponent {
    …
}
```

#1: Define an Interceptor Binding Type

- Define Interceptor Binding Type called “Logged”

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

#2: Write Interceptor Implementation

- Annotate the implementation class with Interceptor Binding type and @Interceptor
- Implement interceptor method

```
@Logged              // Interceptor binding type
@Interceptor
public class LoggedInterceptor implements Serializable {
    private static final long serialVersionUID = 1L;
    public LoggedInterceptor() {
    }

    @AroundInvoke
    public Object logMethodEntry(InvocationContext invocationContext) throws Exception {
        System.out.println(
                "Entering method: " + invocationContext.getMethod().getName() + " in class "
                + invocationContext.getMethod().getDeclaringClass().getName());
        return invocationContext.proceed();
    }
}
```

#3: Apply Interceptor to a target

- Annotate the the target class/method with interceptor

```
@Logged
public String pay() {
        this.setDatetime(Calendar.getInstance().getTime());

        switch (paymentOption) {
        case DEBIT:

            PaymentEvent debitPayload = new PaymentEvent();
            debitPayload.setPaymentType("Debit");
            debitPayload.setValue(value);
            debitPayload.setDatetime(datetime);
            debitEvent.fire(debitPayload);

            break;
      ...

  }
```

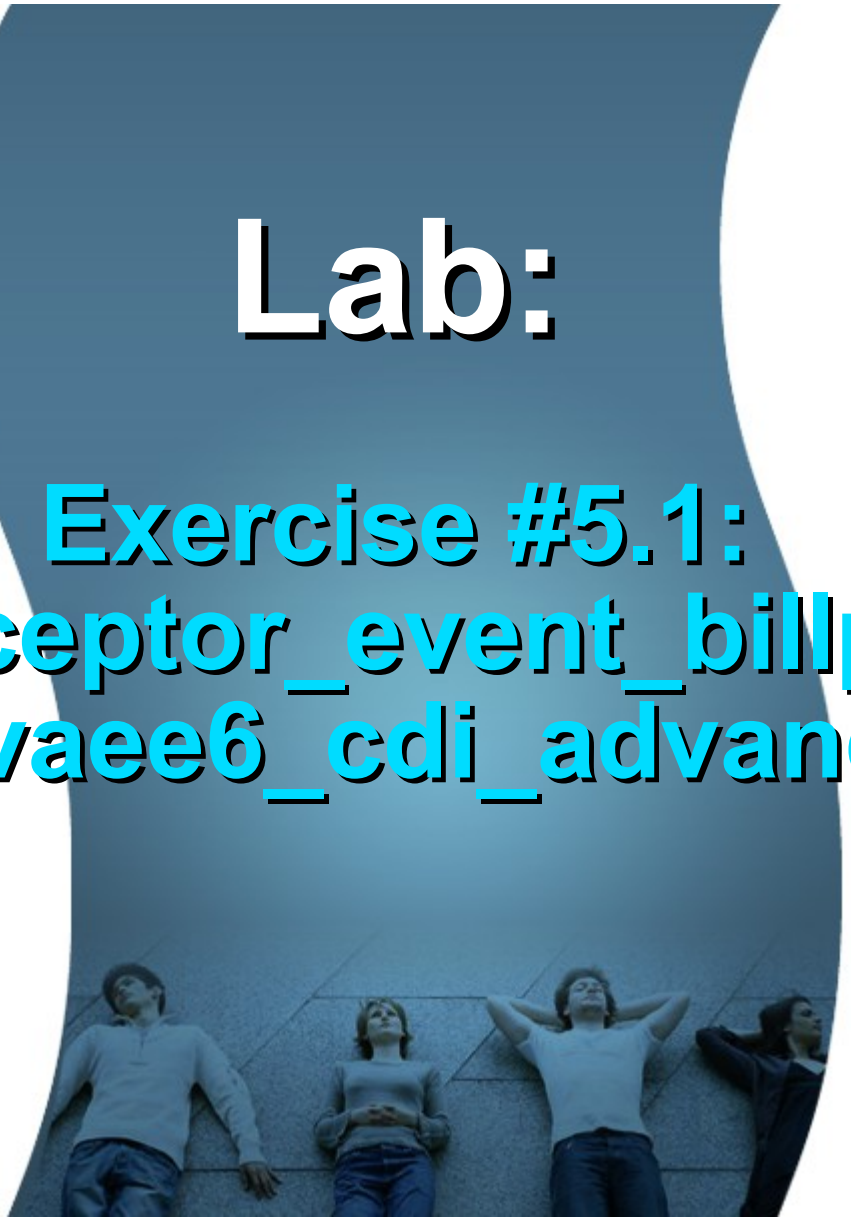
Enable Interceptor

- Interceptor has to be activated through “beans.xml”

```
<beans >
   <interceptors>
      <class>billpayment.interceptor.LoggedInterceptor</class>
   </interceptors>
</beans>
```

Lab:

Exercise #5.1: cdi_interceptor_event_billpayment 4533_javaee6_cdi_advanced.zip



Multiple Interceptors

- Multiple interceptors can be specified

```
@Transactional                    // Interceptor
public class AccountManager {

        @Secure                   // Interceptor
        public boolean transfer(Account a, Account b){

        }

    }
```

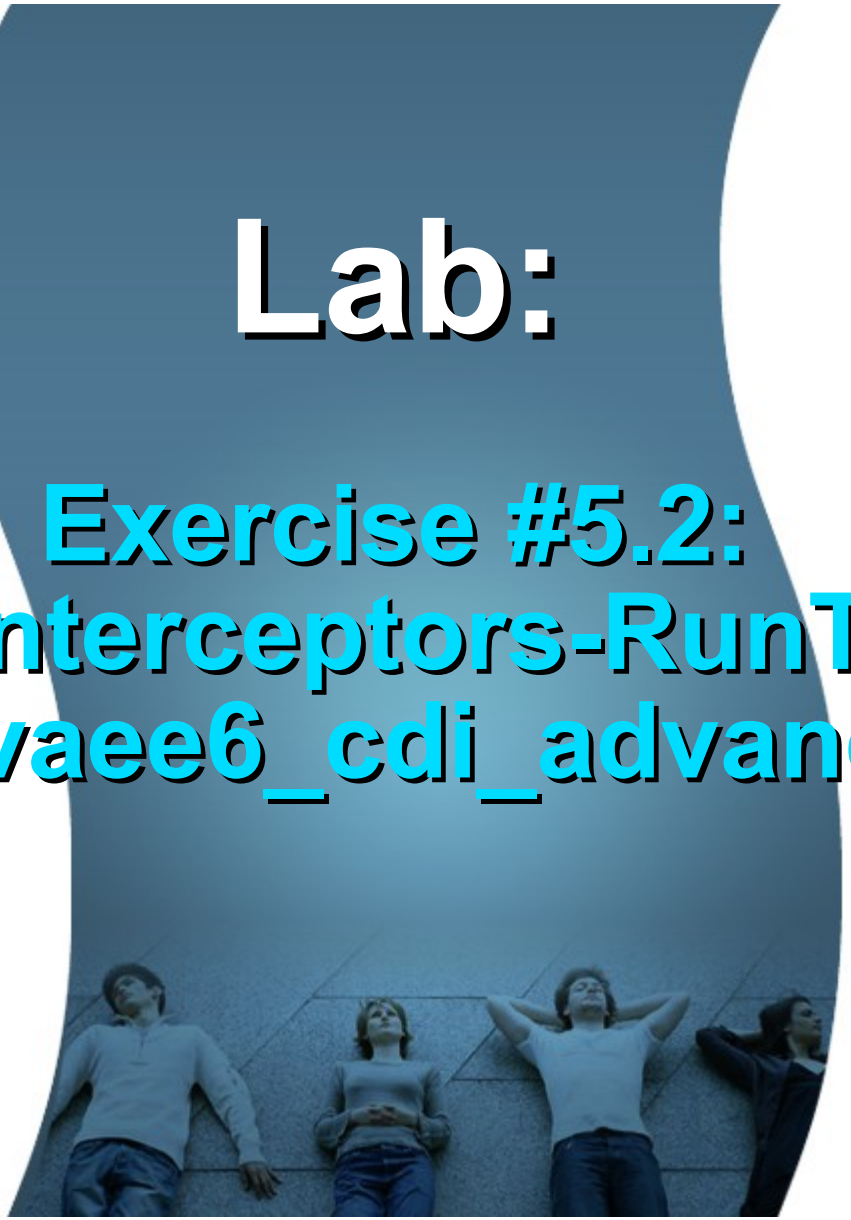
Specify the Order in “beans.xml”

- Ordering is per module
- Interceptors are are applied in order listed in “beans.xml”

```
<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>
```

Lab:

Exercise #5.2: cdi-interceptors-RunTest 4533_javaee6_cdi_advanced.zip



Composite Interceptor Binding

```
// Define composite @BusinessOperation interceptor with
// @Secure and @Transactional interceptors
public
@Secure
@Transactional
@InterceptorBinding
@Retention(RUNTIME)
@Target(TYPE)
@interface BusinessOperation {}
```

Use Composite Interceptor

- User does not know if it is a composite interceptor – another example of loose coupling

```
public
@BusinessOperation
class AccountManager {

    public boolean transfer(Account a, Account b) {
      …
    }

}
```

Decorator

What is a Decorator?

- Decorators implement the Decorator design pattern
  - Allows implementation of an additional business logic for a bean
- A Decorator decorates interfaces they implement
- *@Delegate* is used to inject the original object
  - Original object business logic can be be invoked within the decorator
- Decorators must be activated through “beans.xml”

Define a Decorator

```
@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject
    @Delegate
    @Any
    Coder coder;

    public String codeString(
        String s,
        int tval) {
        int len = s.length();

        // The decorator's codeString method calls the delegate
        // object's codeString method to perform the actual encoding.
        return "\"" + s + "\" becomes " + "\"" + coder.codeString(s, tval)
        + "\", " + len + " characters in length";
    }
}
```

Enable Decorator

- Decorator has to be activated through “beans.xml”

```
<beans >
   <decorators>
      <class>decorators.CoderDecorator</class>
   </decorators>
</beans>
```

Lab:

Exercise #6:
cdi_decorators
4533_javaee6_cdi_advanced.zip

Stereotypes

What is a Stereotype?

- A Stereotype is an annotation that groups other annotations
  - Solves “too many annotations” problem
- A stereotype can group
  - Scope annotation
  - Interceptor binding annotations
  - @Named annotation
  - @Alternative annotation

Without Stereotypes

- Without using stereotypes
  - > Annotations pile up
  - > Duplication

```
public
@Secure
@Transactional
@RequestScoped
@Named
class AccountManager {

    public boolean transfer(Account a, Account b) {
      …
    }

}
```

Without Stereotypes

- Without using stereotypes
  - > Annotations pile up
  - > Duplication

```
public
@Secure
@Transactional
@RequestScoped
@Named
class AccountManager {

    public boolean transfer(Account a, Account b) {
      …
    }

}
```

Define and use Sterotype

- Define a stereotype

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

- Use the sterotype

```
@BusinessComponent
class AccountManager {
    public boolean transfer(Account a, Account b) {
        …
    }
}
```

Built-in Stereotype in JSF: @Model

- Instead of using JSF managed beans, just annotate a bean @Model, and use it directly in your JSF view

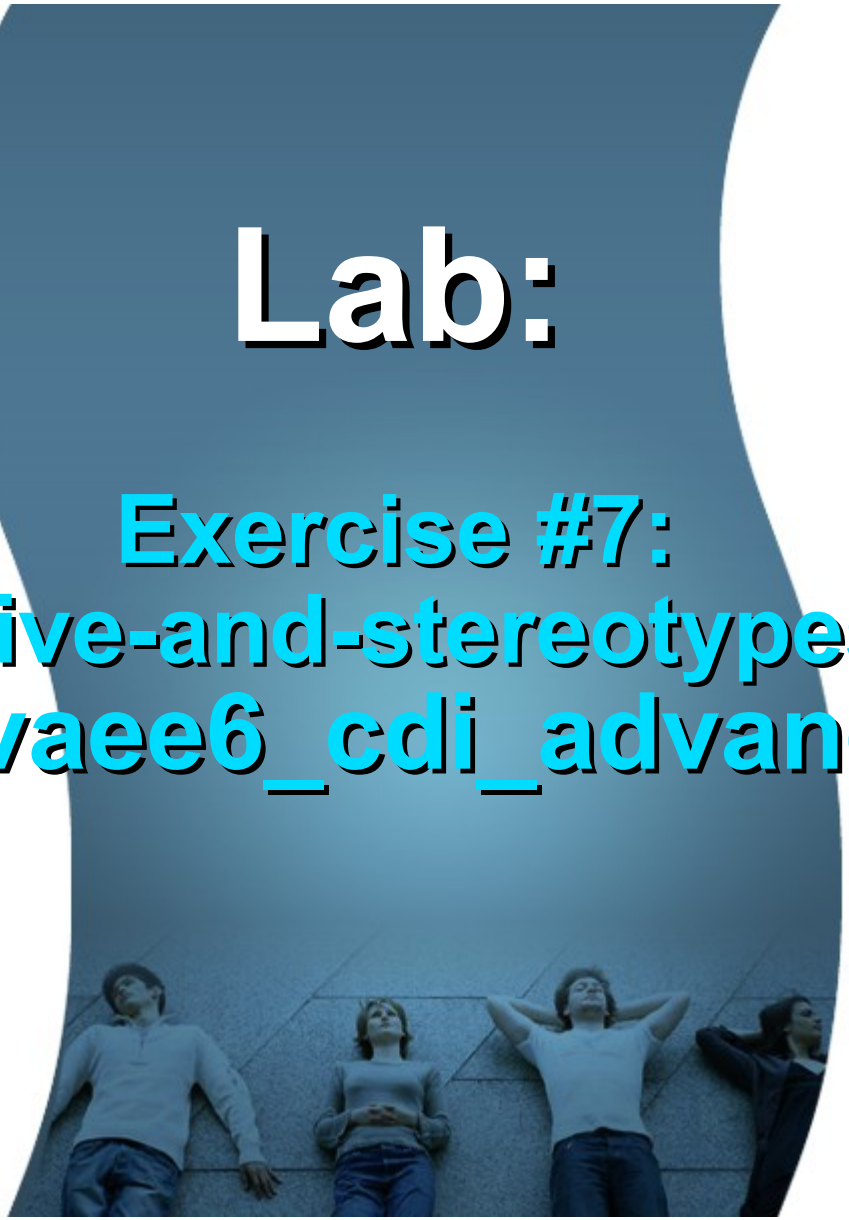
```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

- Use the sterotype

```
@Model
class AccountManager {
    public boolean transfer(Account a, Account b) {
        …
    }
}
```

Lab:

Exercise #7:
cdi-alternative-and-stereotypes-RunTest
4533_javaee6_cdi_advanced.zip



Code with Passion!
JPassion.com

Built-in Qualifiers

- @Named
  - Gives a name to a bean
- @Default
  - If no qualifier is specified, @Default is assumed
- @New
  - Allows to obtain a dependent object of a specified class, independent of the declared scope.
- @Any
  - Declare an injection point without specifying a qualifier

@Default

```
@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;
   }

}
```

```
@Default @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;
   }

}
```

@Any

```
// From time to time, you'll need to declare an injection point
// without specifying a qualifier.

// All beans have the qualifier @Any. Therefore, by explicitly
// specifying @Any at an injection point, you suppress the default
// qualifier, without otherwise restricting the beans that are eligible
// for injection.

//  Useful if you want to iterate over all beans with a certain bean type
@Inject
void initServices(@Any Instance<Service> services) {
   for (Service service: services) {
      service.init();
   }
}
```