Session: 7

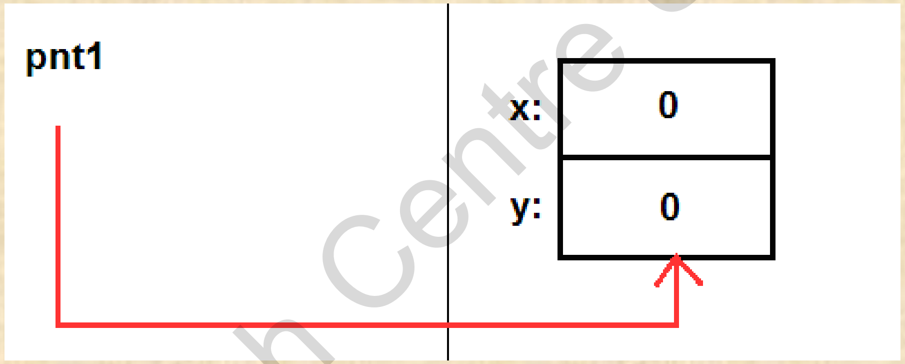
Interceptors and Dependency Injection

Objectives

- Explain aspect-oriented programming
- List the advantages of aspect-oriented programming
- Explain interceptors and their usage in enterprise applications
- Describe the different types of interceptor methods
- Explain how to define the interceptors and their levels of description
- Explain the creation of interceptor class
- Explain how to implement business method and lifecycle callback method interceptors
- Describe various aspects of context dependency and Injection
- Describe how to use CDI in enterprise applications

Introduction to Aspect-Oriented Programming 1-4

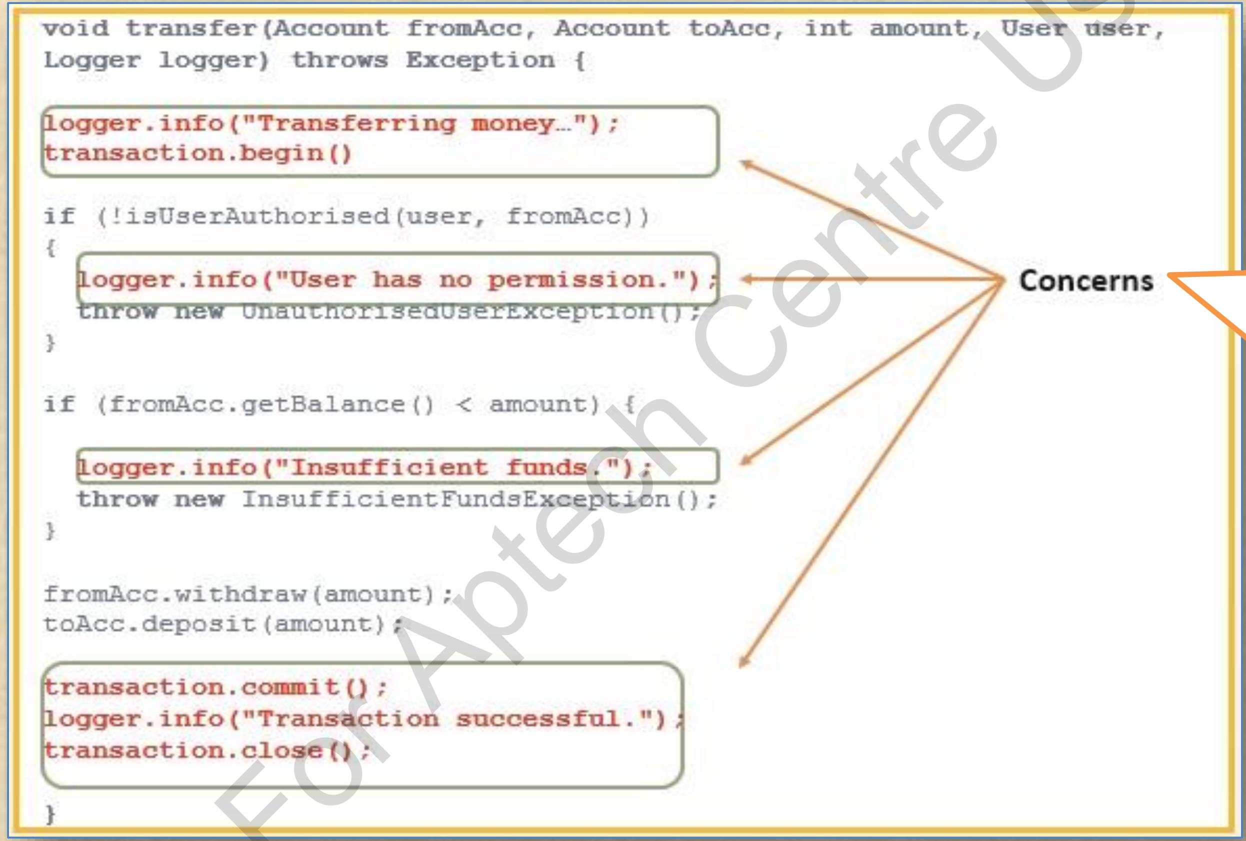
- Every business logic has some dependencies on its execution.
- For example, the parameters passed to a business method must be validated, before they are processed further by the method.



- In a software architecture design, the services provided to the business logics for their successful execution are termed as concerns.

Introduction to Aspect-Oriented Programming 2-4

- Following figure shows the concerns incorporated in the banking application:



```
void transfer(Account fromAcc, Account toAcc, int amount, User user,
Logger logger) throws Exception {

logger.info("Transferring money…");
transaction.begin()

if (!isUserAuthorised(user, fromAcc))
{
   logger.info("User has no permission.");
   throw new UnauthorisedUserException();
}

if (fromAcc.getBalance() < amount) {

   logger.info("Insufficient funds.");
   throw new InsufficientFundsException();
}

fromAcc.withdraw(amount);
toAcc.deposit(amount);

transaction.commit();
logger.info("Transaction successful.");
transaction.close();

}
```

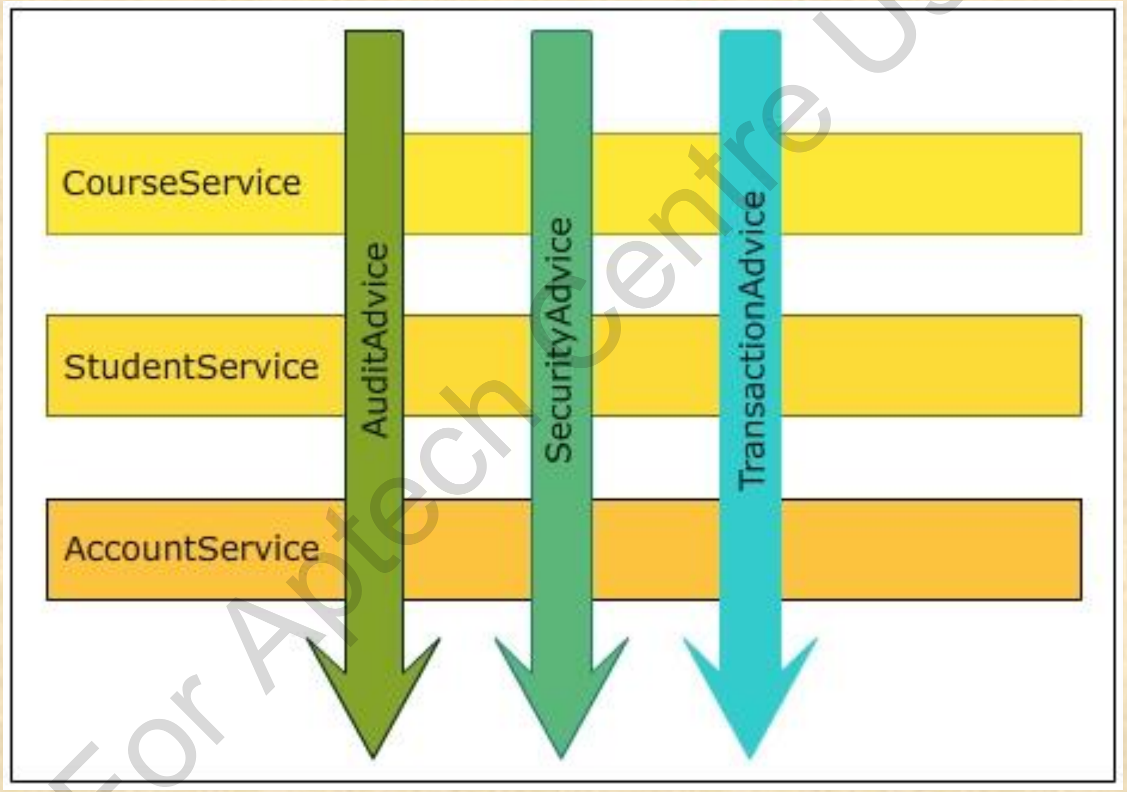
The concerns are interrelated to the business logics and are also referred to as business logic concerns.

Introduction to Aspect-Oriented Programming 3-4

- To handle these types of concerns separately from the business logic, modern languages and frameworks provide support for Aspect-Oriented Programming (AOP).
- AOP allows the code developer to express cross-cutting concerns into stand-alone modules called aspects.
- The aspects intercept the request invocation executed for the object.
- Then, the aspects are injected into the functionality and finally, the request is delegated to the targeted object.
- Some of the common aspects include logging, transaction, security, connection, and so on.

Introduction to Aspect-Oriented Programming 4-4

- Following figure shows the aspects injecting into various business codes of the enterprise application:



Advantages of Aspect-Oriented Programming 1-2

Reusability

- By modularizing the code, it can be injected into multiple objects.

Separation of concern

- Separation of concerns from the business logic makes the implementation independent of the concerns.

Focus

- Developers can concentrate on aspects separately without merging them with the business logics.

Advantages of Aspect-Oriented Programming 2-2

On Java EE platform, EJB 3 supports AOP framework through Interceptors.

- **Interceptors**
  - Interceptors are used to enable inclusion of aspects within the EJBs.
  - They provide the capability to intercept business methods and lifecycle callback methods.

Interceptors 1-2

- Allow the developers to interpose EJB business methods.
- Add a wrapper code which is executed before or after the method is called.
- Are used to perform tasks such as logging information of bean method execution, auditing, and so on.
- Provide a fine grained control to the bean developer on the lifecycle methods of a bean.
- Used with Session beans and Message-driven beans.
- Defined through annotations or deployment descriptors.

Interceptors 2-2

- Following are some of the scenarios in which interceptors can be executed:
  - Method parameters are to be validated, before they are passed to the bean method.
  - Security checks are required to be performed, before executing the bean method.
  - Logging or profiling operations are performed, during the bean method execution.
  - Certain operations required to be performed, before or after class instantiation in the application.

Types of Interceptors

- **Business method interceptors**
  - Also referred as '`AroundInvoke`' interceptors.
  - Intercepts invocation of business methods in Session/Message-driven beans.
  - Annotated with `javax.interceptor.AroundInvoke` annotation which designate the business method as an interceptor method.
- **Lifecycle callback interceptors**
  - Intercepts the invocation of lifecycle methods of the session beans or message-driven bean instances.
- **Timeout callback interceptors**
  - Also referred as '`AroundTimeOut`' methods.
  - Defined for EJB timer services.

Defining Interceptors

- Developers can define interceptors in one of the following two ways:

As an interceptor method within a target class

- Target class is the bean class.
- Methods are annotated with `@javax.Interceptor.Interceptors` annotation.
- Bean can either impose interceptor class on all methods or impose interceptor class on explicit methods.

As a separate interceptor class

- Interceptor class contain methods which can be invoked along with the lifecycle callback methods of the target class or business methods of the target class.
- Only one interceptor class can be defined for a target class.

Levels of Interceptors

- Interceptors are defined at different levels in the application:
  - Default interceptors
  - Class-level interceptors
  - Method-level interceptors

Default Interceptors

- They are defined in the deployment descriptor.
- They are applicable across all the EJBs deployed in the application.
- Following code snippet shows `ejb-jar.xml` with default interceptors:

```
<ejb-jar
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd” version="3.1">
<assembly-descriptor>
 <interceptor-binding>
   <ejb-name> * </ejb-name>
   <interceptor-class>com.beans.MyInterceptor</interceptor-class>
 </interceptor-binding>
...
</assembly-descriptor>
```

Class-Level Interceptors

- They are defined on all methods of bean class.
- They are used to define the interceptors for all methods in the class using `@Interceptors` annotation.
- Following code snippet shows the usage of class-level interceptors:

```
@Stateless
  @Interceptors(value=com.beans.MyInterceptor.class)
   public class ApplicationBean {
          . . .
  }
```

Method-Level Interceptors

- They are defined to intercept a method invocation in the bean class.
- `@Interceptors` annotation is used to define method-level interceptors.
- Following code snippet demonstrates the usage of method-level interceptors:

```
. . .
@Interceptors({AccountsConfirmInterceptor.class})
   public void sendBookingConfirmationMessage(long
orderId)
   {
      ...
   }
 .. .
```

Configuring an Interceptor Class

Every Interceptor class is associated with a Stateless or Stateful session bean.

Following are the steps in configuring an interceptor class:

- Create an interceptor class as a simple POJO class.
- Define an interceptor method within the class.
- Designate the method with `@AroundInvoke` annotation or lifecycle annotation.
- Associate the interceptor class with EJB Session bean.

Creating Business Method Interceptor 1-6

- `javax.interceptor.AroundInvoke` annotation is used to create business method interceptors.
- Following code snippet shows the signature of the business interceptor method annotated with `@AroundInvoke` annotation:

```
@AroundInvoke
Object <METHOD_NAME>
(javax.Interceptor.InvocationContext) throws Exception
{
   .. .
}
```

- The method designates as the interceptor method.
- The developer can use only one `@AroundInvoke` annotation per class.

Creating Business Method Interceptor 2-6

- The advantage of interceptors is that they provide the developer a way to add functionalities on the business methods without modifying the method code.
- Interceptor methods can be declared `public`, `private`, `protected`, or `package` level access.
- They cannot be declared `final` and `static`.
- They may throw runtime exceptions.
- They are invoked with `InvocationContext` object as a parameter.

Creating Business Method Interceptor 3-6

- **InvocationContext object**
  - Is the generic representation of the business method for which the interceptor is defined.
  - Includes information such as target bean instance on which the interceptor is invoked, parameters with which the target bean is invoked, and so on.

Creating Business Method Interceptor 4-6

- Following are some of the methods of `InvocationContext`:
  - `Object getBean()`: Returns a reference to the bean on which the method is invoked.
  - `Method getMethod()`: Returns a reference to the invoked method.
  - `Object[] getParameters()`: Returns the parameters passed to the method.
  - `void setParameters(Object[] parameters)`: Sets the parameters of the object.
  - `Map getContextData()`: Get contextual data that can be shared in a chain.
  - `Object proceed()`: Proceeds to the next interceptor in the chain or the business method, if it is the last interceptor.

Creating Business Method Interceptor 5-6

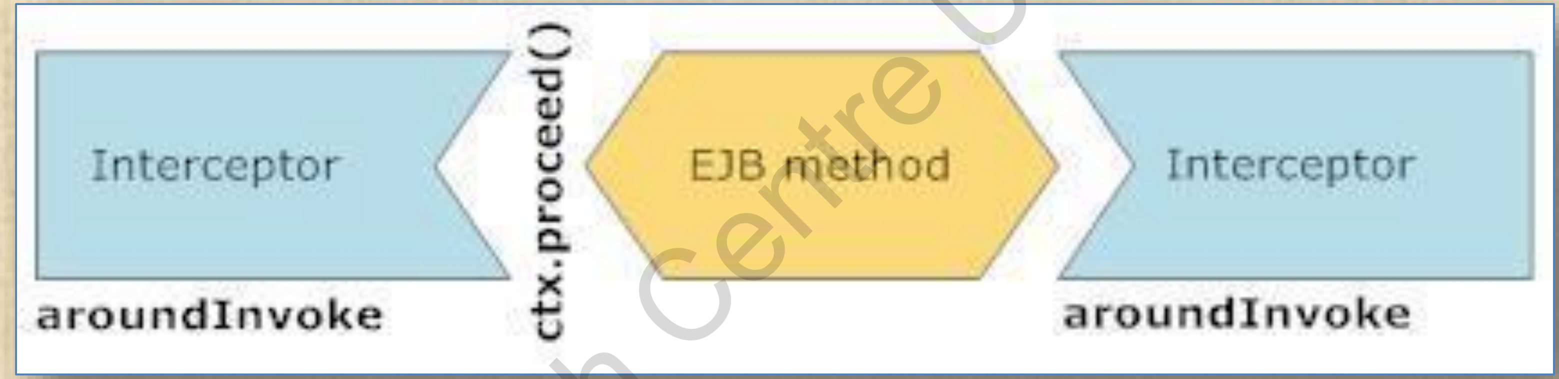
- Following code snippet demonstrates the definition of an interceptor class:

```
. . .
public class MyInterceptor {
@AroundInvoke
public Object businessIntercept(InvocationContext ctx)
throws Exception {
   Object result = null;
   System.out.println("perform intercepting operations");
   try {
       result = ctx.proceed();
        return result;
   } finally { . . .
           }
 }
}
```

- The reference of `InvocationContext`, `ctx` is obtained.
- The method `proceed()` is invoked on the ctx reference to get the next configured interceptor in the chain.
- If another interceptor has to be invoked as a part of method call, then `proceed()` invokes the `@AroundInvoke` method of the other interceptor.

Creating Business Method Interceptor 6-6

- Following figure demonstrates wrapping business method with a chain of interceptors:



Applying Interceptor 1-5

- Developer can apply interceptors to the bean class by configuring the deployment descriptor.
- Developers can also use `@Interceptors` annotation to apply the interceptor.
- Following is the syntax for applying the interceptors:

```
@Interceptors({Interceptor1.class,Interceptor2.class})
targetMethod(){
 . . .
}
```

Applying Interceptor 2-5

- Following code snippet shows the code of a target class which uses interceptor:

```
. . .
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptors;
import javax.interceptor.InvocationContext;

@Stateless
@LocalBean
public class ExampleBean {
@Interceptors(value = interceptors.MyInterceptor.class)
public void sayHello(){
  System.out.println("SayHello");
}
```

- The code shows the target class on which the interceptor class is defined.
- The **BusinessIntercept()** method is invoked before any method of the **ExampleBean** class is invoked.

Applying Interceptor 3-5

- Following code snippet shows the invocation of a bean method on which interceptor is defined:

```
package Beans;
public class InterceptorDemo {
  public static void main(String[] args) {
       ExampleBean  E = new ExampleBean();
        E.sayHello();
    }
}
```

- The code instantiates the `EmployeeBean` class and invokes the method `sayHello()` on it.

Applying Interceptor 4-5

- While defining the interceptor as an interceptor class, it must meet the following requirements:
  - The interceptor class must have a default public constructor.
  - The interceptor classes should be defined with `@Interceptors` annotation.
  - When a target class has multiple interceptor classes defined for it, the order of Interceptor invocation should be defined through annotations.
  - The order of invoking the interceptors in case of multiple interceptors can be defined in the deployment descriptor of the application also.
  - The order defined in the deployment descriptor overrides the order defined through annotations.

Applying Interceptor 5-5

- Following code snippet shows the code to specify the interceptors in the deployment descriptor for a particular method:

```
...
<interceptor-binding>
<target-name> Target bean class </target-name>
<interceptor-class>First interceptor class </interceptor-class>
<interceptor-class>Second interceptor class</interceptor-class>
<interceptor-class>Third interceptor class</interceptor-class>
<method-name>Method to be intercepted</method-name>
</interceptor-binding>
...
```

- The order in which the interceptors are invoked is same as the order in which these interceptors are specified in the deployment descriptor.

Life Cycle of an Interceptor

- Life cycle of an interceptor is the same as the target class or target method.
- When an instance of target class is created, then the interceptor class is also instantiated for the target class.
- In case of multiple interceptor classes, then an instance of each interceptor class is created.

Lifecycle Callback Interceptors 1-3

- Lifecycle callback events occur whenever the bean object changes from Does Not Exist state to Ready state or when the bean object transits from activated state to passivated state, and so on.
- The lifecycle callback events are handled through the lifecycle callback methods.
- Following are the annotations related to the lifecycle methods of the bean class:
  - `javax.interceptor.AroundCosntruct`
  - `javax.annotation.PostConstruct`
  - `javax.annotation.PreDestroy`

Lifecycle Callback Interceptors 2-3

- Apart from this, Stateful Session bean also have methods annotated with `@PrePassivate` and `@PostActivate`.
- The syntax for declaring the lifecycle interceptor methods is as follows:
  - `@callback-annotation`
  - `void method-name(InvocationContext ctx);`
- The method defined in the interceptor class will be annotated with the callback annotation.
- The return value of these method is void, as EJB callback methods do not return any value.

Lifecycle Callback Interceptors 3-3

- Following code snippet shows an interceptor class for a Stateful Session bean with lifecycle callback interceptor methods:

```
public class MyStatefulSessionBeanInterceptor {
protected void myInterceptorMethod (InvocationContext ctx) {
   . . .
      ctx.proceed();
    . . .
    }
@PostConstruct
@PostActivate
protected void myPostConstructInterceptorMethod(InvocationContext ctx) {
...    ctx.proceed();   ...
    }
@PrePassivate
 protected void myPrePassivateInterceptorMethod (InvocationContext ctx)
{     ...
        ctx.proceed();
        ...
    }
}
```

Timeout Interceptors

- Timeout interceptors are invoked when a timeout event occurs on an object.
- `@AroundTimeOut` annotation is prefixed for such interceptor methods.
- Following is the syntax for using `@AroundTimeOut` annotation:

```
@AroundTimeOut
public Object InterceptorMethod(){
. . .
}
```

Interceptor Chaining 1-2

- A bean class can have multiple interceptors defined on it.
- The order of executing the interceptors has to be defined in the deployment descriptor.
- There are certain rules to determine the order for executing multiple interceptors.

Interceptor Chaining 2-2

Following are the rules which apply to interceptor chaining:

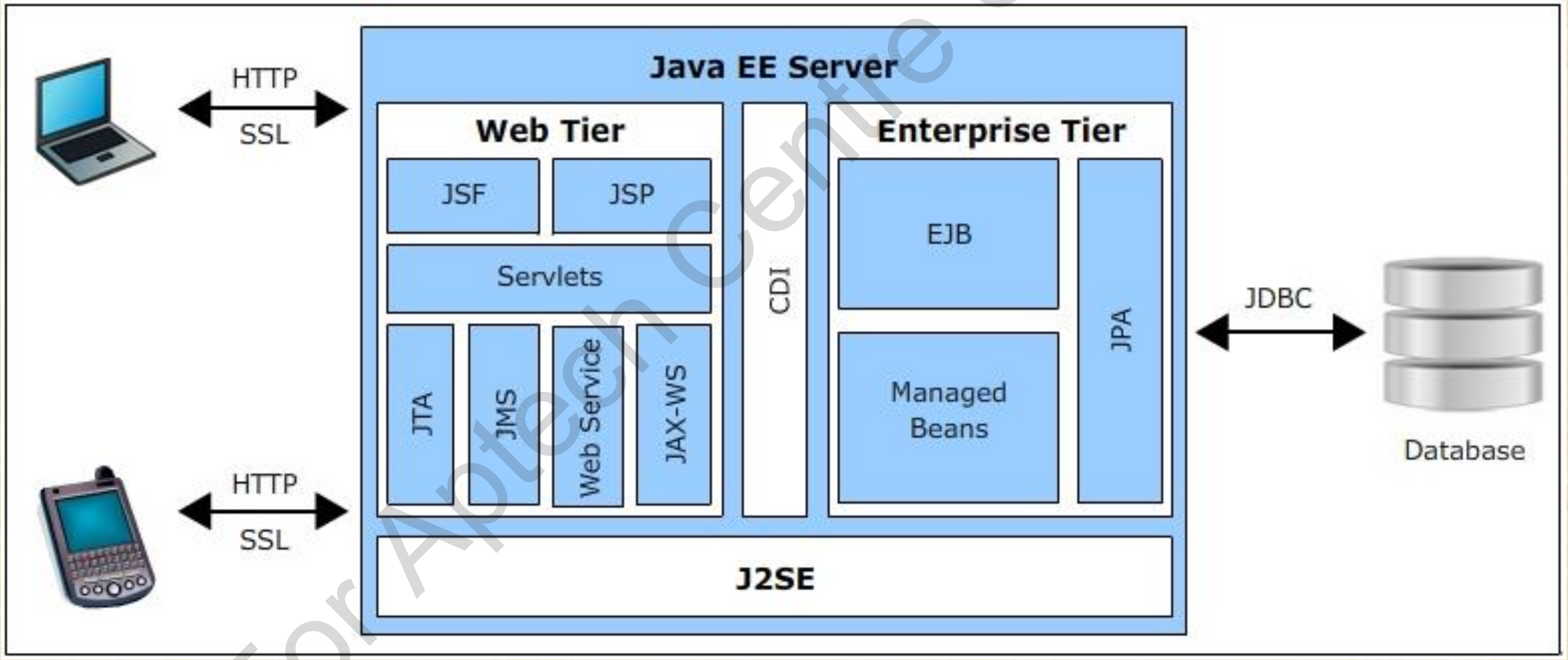
- The target class has a set of default interceptors which are defined in the deployment descriptor.
- The order of execution is defined through `@Interceptors` annotations.
- When interceptors are defined as a hierarchy of classes, the interceptor invocation starts from the super class.
- Interceptors can be associated with a priority through `javax.annotation.priority` annotation.
- Interceptors annotated with `AroundConstruct` are executed before the target method.

Context and Dependency Injection (CDI) 1-3

- CDI is a standard used to define the structure of the application.
- CDI enables integration of various components of the application in a loosely coupled manner.
- CDI allows EJBs to be used as managed beans for JSF applications.
  - This helps the Web tier to interact directly with the business and persistence tiers.
- CDI is also used to resolve dependencies among managed beans.

Context and Dependency Injection (CDI) 2-3

- Following figure shows the interrelationships among different specifications in Java EE using CDI:



Context and Dependency Injection (CDI) 3-3

Following are the services provided by the CDI in Java EE:

- Provides contexts for the lifecycle of Stateful Session bean.
- Provides dependency injection which allows resources to be injected into the application in a type-safe way.
- Uses expression language for integrating application components.
- Allows associating interceptors with business methods.
- Provides Service Provider Interface (SPI) to integrate third party frameworks into Java EE.
- Decouples client and server through well-defined basic datatypes or object types.
- Decouples the life cycle of the collaborating components.

Implementing Managed Beans 1-3

All the managed beans are implemented as Java classes.

Following are the requirements of managed bean classes:

- Should not be non-static inner class.
- Should be a concrete class or should be annotated with `@Decorator`.
- Should not be annotated with an EJB component defining annotations.
- Should have a default constructor.
- Should declare a constructor annotated with `@Inject`.

Implementing Managed Beans 2-3

- Following code snippet demonstrates the usage of `@Inject` annotation:

```
. . .
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.inject.Inject;
@Stateless
@LocalBean
public class InjectBeanDemo {
@Inject
private  HelloBean H;
public InjectBeanDemo(){
    H= new HelloBean();
}
public static void main(String args[]){
    InjectBeanDemo I = new InjectBeanDemo();
} }
```

A **HelloBean** is injected into the current bean through `@Inject` annotation.

Implementing Managed Beans 3-3

- Following code snippet shows the `HelloBean` which is injected in `InjectBeanDemo`:

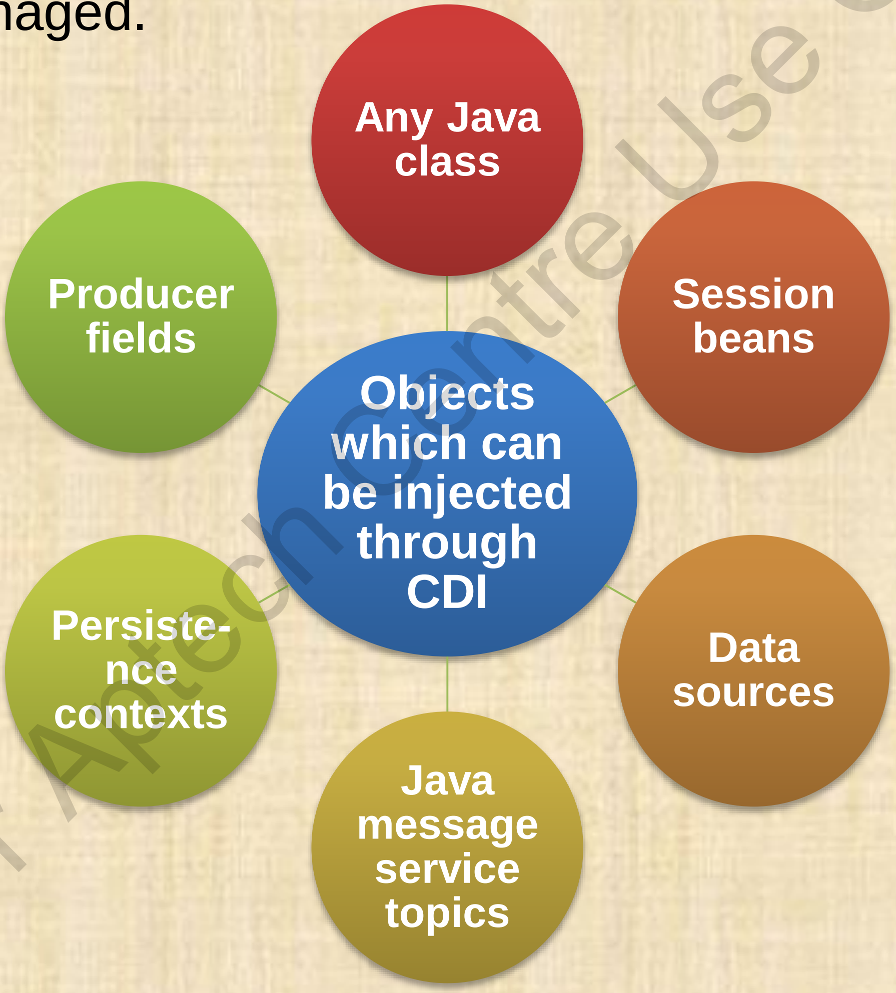
```
package beans;

import javax.ejb.LocalBean;
import javax.ejb.Stateless;
@Stateless
@LocalBean
public class HelloBean {

    public HelloBean() {
         System.out.println("In Hello Bean");
    }
 }
```

Beans as Injectable Objects

- CDI makes it possible to inject objects into applications which are not container-managed.



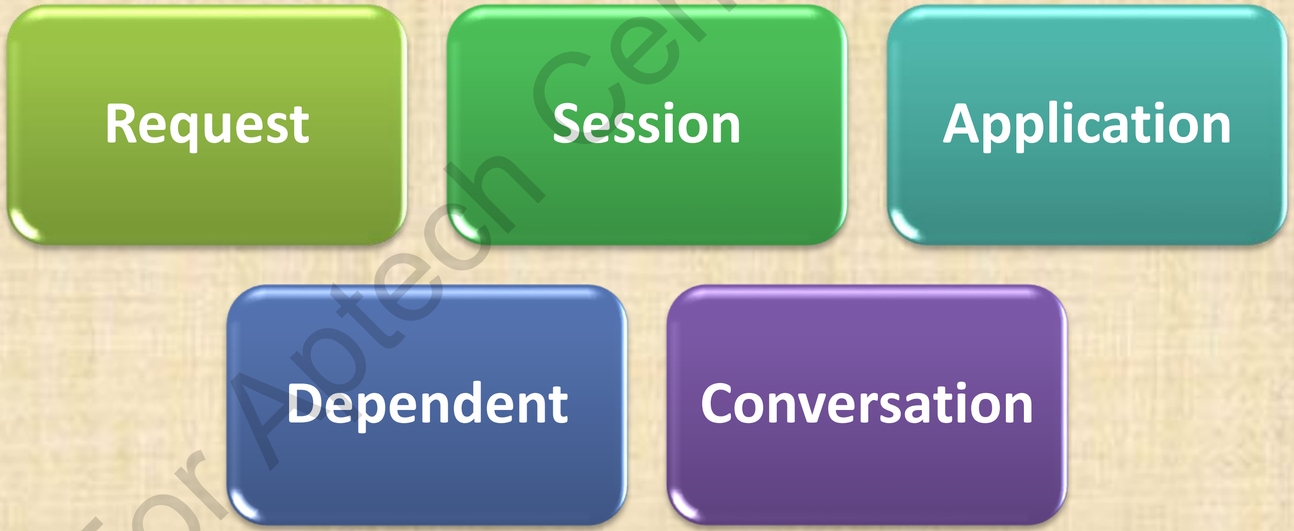
Injecting Beans

- Enterprise beans are injected as resources into other bean classes.
- Enterprise beans are injected using the annotation `javax.inject.Inject`.
- Following code snippet demonstrates the process of injecting `CustomerInformation` into `LoanApproval` bean:

```
import javax.inject.Inject;
class LoanApproval{
@Inject CustomerInformation;
  . . .
}
```

Configuring a CDI Application

- To configure a CDI application, the scope of the bean has to be appropriately defined.
- The scope of a session bean can have any one of the following values:



Packaging CDI Applications 1-2

- All the beans managed by CDI are packaged as bean archive files.
- There are two variants of bean archive files:
  - Implicit bean archive files
  - Explicit bean archive files
- Explicit bean archive comprises `beans.xml` deployment descriptor.
- Implicit bean archive is one which contains some beans annotated with a scope type.

Packaging CDI Applications 2-2

- Following code snippet shows `beans.xml` deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

- The container checks all the Java classes in the application for bean defining annotation and discovers the beans through them.

Summary

- Interceptors are invoked when either business methods are invoked or when bean objects are instantiated.
- Interceptors can be implemented as classes or methods.
- When interceptors are defined as a class then interceptor method has an `InitialContext` parameter.
- There can be more than one interceptor defined for the bean class.
- Multiple interceptors can be defined using `@Interceptors` annotation or through deployment descriptor.
- CDI is used in enterprise applications for type-safe dependency injection.
- To configure a CDI application, the scope of the bean has to be appropriately defined for the bean class.