Session: 1

Introduction to Jakarta Enterprise Beans

Jakarta Enterprise Beans in Action

Objectives

- Define an overview of Jakarta Enterprise Beans
- Describe an Enterprise Bean
- Explain the necessity for Enterprise Beans
- Describe how to set up the environment for Jakarta Enterprise Beans

Introduction

- Jakarta Enterprise Beans:
  - Was earlier known as Enterprise JavaBeans or EJB
  - Is an API in the Jakarta EE platform for enterprise applications
  - Is used as a server-side software component for building scalable enterprise software
  - Is used for creating distributed, transactional, secure, and portable Jakarta EE applications
  - Describes how business logic can integrate seamlessly with application server's persistence services
  - Is extremely adaptable and simple to set up
  - Any Jakarta EE-compatible runtime can be used with it such as `GlassFish`, `Payara`, or `WildFly`

What is an Enterprise Bean?

A server-side component in enterprise applications

Written in Java and annotated using EJB annotations.

Packaged and deployed using JAR files or WAR files.

May be able to describe an infinite variety of business interfaces.

Can be used to build business logic.

The implementation server delivers benefits such as privacy and security, transaction data, and parallel processing monitoring.

Characteristics of an Enterprise Bean 1-2

A container manages the instances of an Enterprise Bean at program execution.

During deployment, an Enterprise Bean can be personalized by modifying its environment configuration.

Characteristics of an Enterprise Bean 2-2

An Enterprise Bean relies on a provider that can only be implemented in a container, which endorses it.

Multiple service data could be stipulated alongside the business logic of the Enterprise Bean.

A client view of an Enterprise Bean is defined by the bean provider.

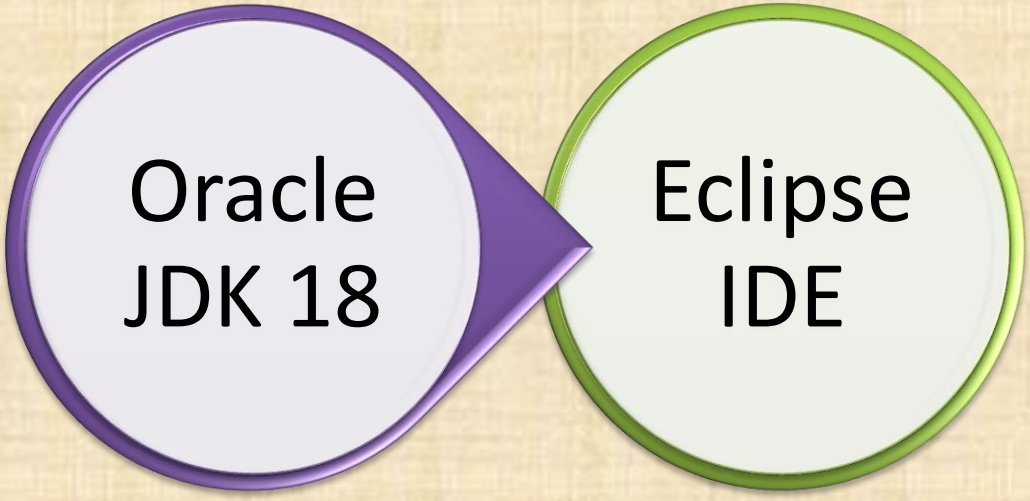
Necessity of Enterprise Beans

- They make it easier to create large, distributed applications.
- System-level services are handled by the EJB container.
- Client developer is not required to write code to implement the business rules.

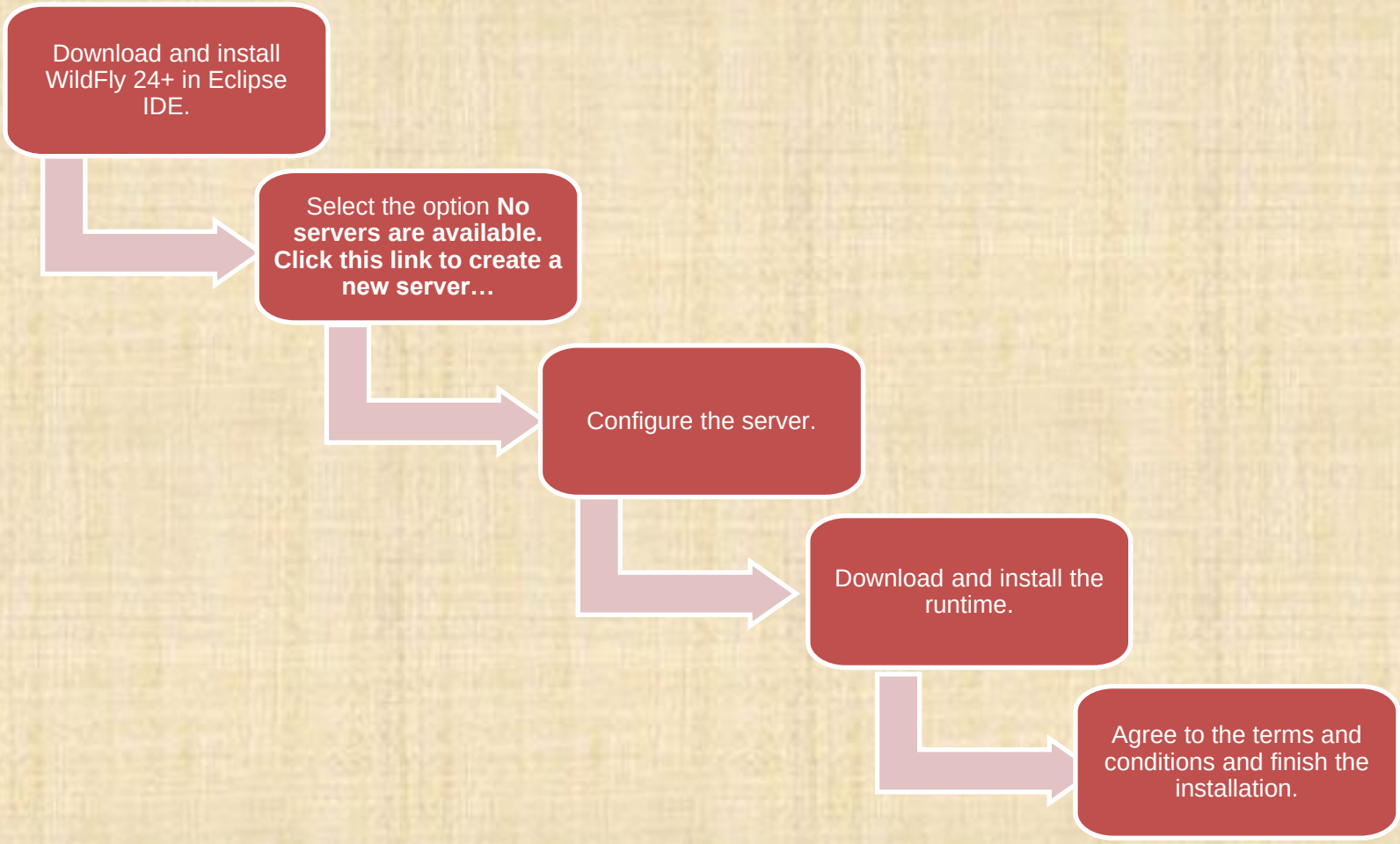
- New applications can be created by combining existing beans.
- Scalability and integrity can be achieved.
- The application that is created using Enterprise Beans will serve a wide range of clients.

Setting Up the Jakarta Enterprise Beans Environment 1-2

Pre-requisites to be installed before setting up the Enterprise Beans Environment



Setting Up the Jakarta Enterprise Beans Environment 2-2



Summary

- Jakarta Enterprise Beans (EJB) is a server-side software component that contains the business logic of an app.
- Enterprise beans offer many benefits to enterprise applications.
- Developers can use any Jakarta EE-compatible runtime such as GlassFish, Payara, or WildFly.
- WildFly is a popular pick for enterprise-capable application designers and users all over the world.
- WildFly, previously known as JBoss Application Server, is a Jakarta EE application server that is open source. Its main goal is to offer a collection of necessary tools for enterprise Java applications.

Session: 2

Session Beans and Their Types

Jakarta Enterprise Beans in Action

Objectives

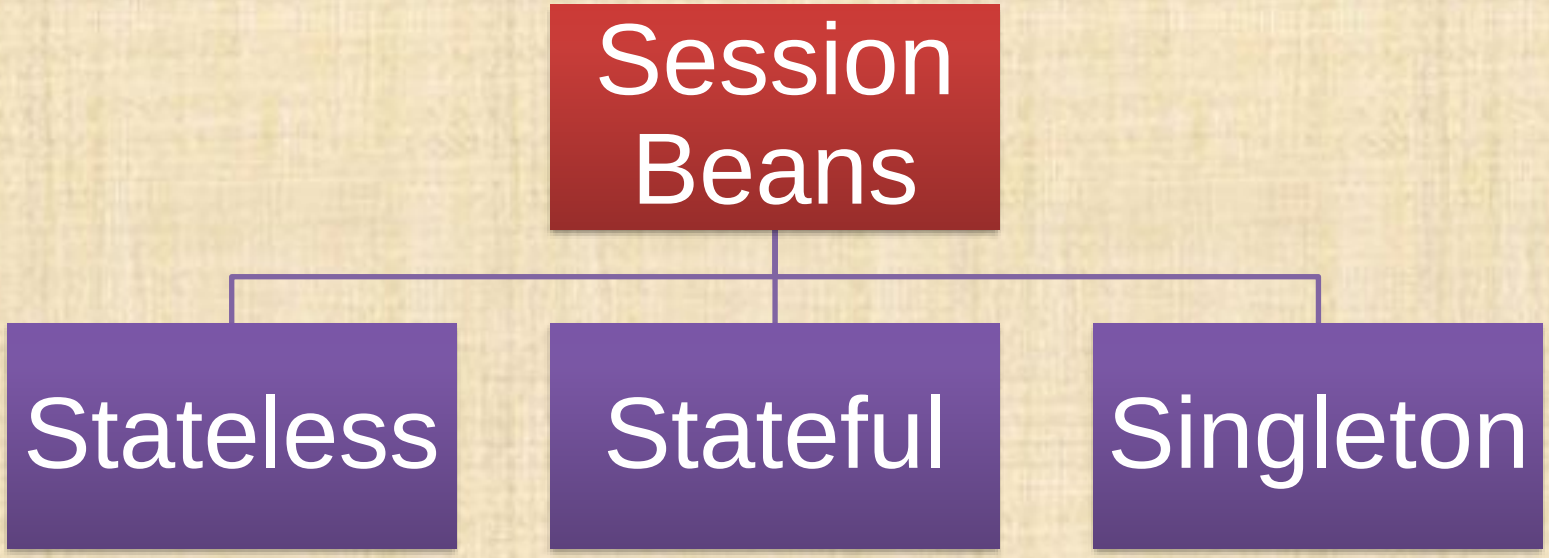
- Elaborate the types of Session Beans: Stateful, Stateless, and Singleton
- Distinguish between Stateful, Stateless, and Singleton sessions
- Describe Entity Bean
- Explain message-driven Beans
- Explain message-driven controlled delivery: delivery active and delivery groups

Introduction to Session Beans

- A Session Bean is a Jakarta component that includes business logic and implements a specific application task.
- Session Beans can be invoked over local or remote Beans.
- The `get/set` methods used by the Session Beans are used to abstract the difficult business logic from the end user.
- Clean and durable code files are made.
- Data for Session Beans is not stored in any database.
- Session Beans are used to establish Web services and increase application performance.

Types of Session Beans

There are three types of Session Beans:



Introduction to Stateful Beans

In Stateful Beans, clients interact with the Bean known as conversational state.

Stateful Beans facilitate only one client at a time. When the client closes, the Session Bean also closes.

The state of the Bean is maintained throughout the session.

The class for the Stateful beans is `@Stateful` tag.

Stateful Beans Creation 1-2

The first step is to create a Stateful session bean in Eclipse IDE.

In the Session Bean class, add Code Snippet 1, which is a program to add two numbers.

Code Snippet 1:

```
package org.stateful.co;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
@Stateful
@LocalBean
public class DemoExampleBean implements
DemoExampleBeanRemote
{
        public int add(int a, int b) {
                return a + b;
        }
}
```

Stateful Beans Creation 2-2

Code Snippet 2 shows the code to be added to the remote interface for `DemoExampleBean`.

The annotation `@Remote` indicates that these methods can be accessed only by a remote client.

Code Snippet 2:

```
package org.stateful.co;
import jakarta.ejb.Remote;
@Remote
public interface DemoExampleBeanRemote {
        public int add(int a, int b);
}
```

Stateful Beans Execution of Remote Client

The first step is to create an application Client project.

An EJB cannot perform any function unless it is invoked by a client.

Code Snippet 3:

```
import org.stateful.co.DemoExampleBean;
import org.stateful.co.DemoExampleBeanRemote;
public class Main {
public static DemoExampleBeanRemote
DemoExampleBean;
public static void main(String[] args) {
DemoExampleBeanRemote DemoExampleBean = new
DemoExampleBean();
System.out.println("Result:"+DemoExampleBean.add(4,
2));
}
}
```

Introduction of Stateless Beans

Stateless Beans only represent business logic. They lack a state. When a stateless bean interacts, it does not save a state.

Stateless Beans have business logic, but does not have data as there is no state.

Stateless Session Beans can provide high flexibility for applications. A Stateless Session Bean can incorporate a Web service, but a Stateful Session Bean cannot.

Stateless Bean Creation 1-2

Code Snippet 4:

```
package org.stateless.ejb;
import java.util.ArrayList;
import java.util.List;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
/**
* Session Bean implementation for class StateLess
*/
@Stateless
@LocalBean
public class StateLess implements StateLessRemote {
  List<String> products;
  public void addProduct(String productName) {
   products = new ArrayList<String>();
   products.add(productName);
  }
  public List<String> getProducts() {
  return products;
  }
}
```

Stateless Bean Creation 2-2

Code Snippet 5:

```
package org.stateless.ejb;
import java.util.List;
import jakarta.ejb.Remote;
@Remote
public interface StateLessRemote
{
 void addProduct(String
productName);
 List getProducts();
}
```

Code Snippet 6:

```
import java.util.ArrayList;
import java.util.List;
import
org.stateless.ejb.StateLessRemote;
public class Main {
  public static StateLessRemote
StateLess;
  public static void main(String[]
args) {
      StateLessRemote StateLess= new
org.stateless.ejb.StateLess();
     List PList = new ArrayList();

StateLess.addProduct("MobilePhone");
     PList = StateLess.getProducts();
    System.out.println(PList);
   }
}
```

Introduction to Singleton Beans

- Singleton Bean is a continual Bean that is called once per application.
- It is used when multiple clients share the same session Bean at the same time.

- A Singleton Session Bean is created once for each application and appears to exist for the entire duration of the application's life cycle.
- It is used in application initialization, Web service endpoint, and as a cleanup tool also.

Difference Between Stateful, Stateless, and Singleton Sessions

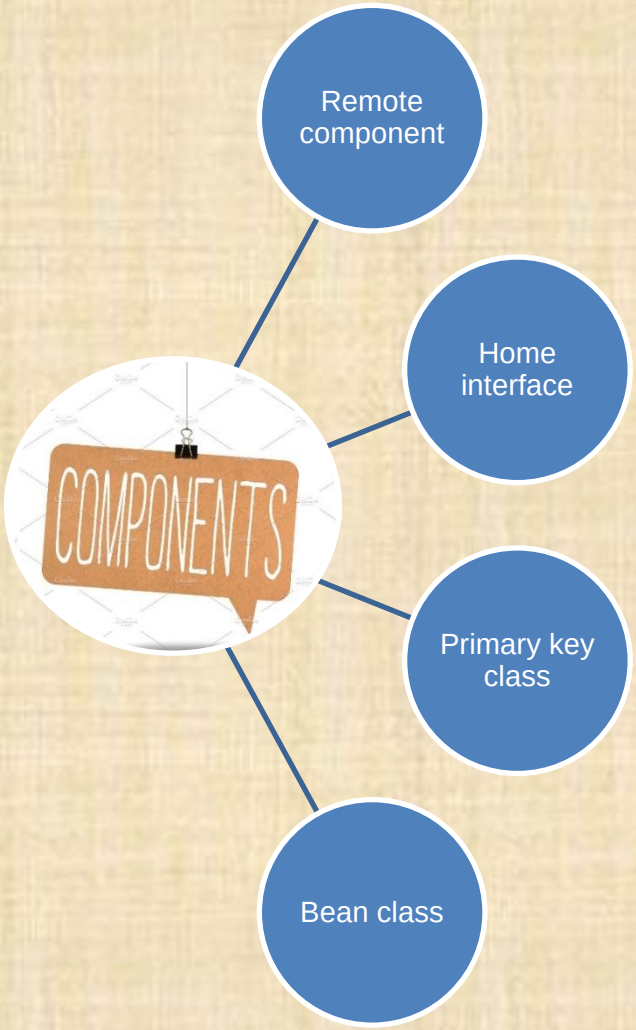
- Stateless Session Bean is a business object without a state (data) that explains the business rules, whereas a Stateful Session Bean does have a state (data) that characterizes the business logic.
- In the Stateless Session Bean, the container does not preserve the conversational state between multiple method calls whereas, in the Stateful Bean, the container preserves the conversational state.
- Singletons are similar to Stateless Session Beans in that they offer comprehensive measurement, but they differ in that there is only one Singleton Session Bean per application.

Entity Bean 1-2

An Entity Bean is a remote object which maintains persistent data, executes complicated business logic may use two or more dependent Java objects, and is uniquely determined by a primary key.

Entity Bean communicates with other Entity Beans by establishing relationships as they hold primary keys.

Components of Entity Beans

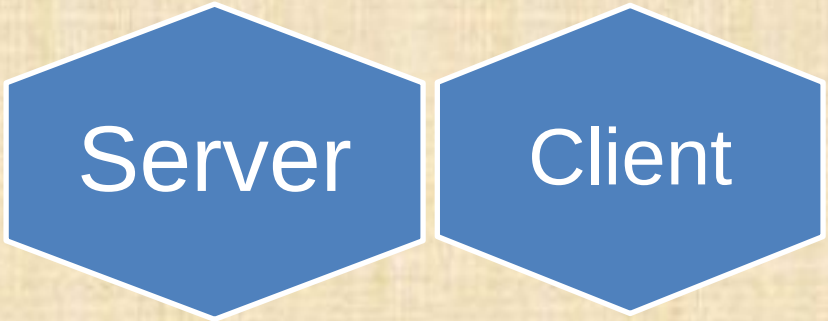


Entity Bean 2-2

Code Snippet 7:

```
@Entity
public class Flight implements Serializable {
Long NUM;
@NUM
public Long getNUM () { return NUM; }
public void setNUM (Long id) { this. NUM =
NUM; }
}
```

Components of Entity Beans



Message-Driven Beans

A Message-driven Bean is an Enterprise Bean that enables asynchronous message handling in Jakarta EE application forms.

One such Message-driven Bean is used as a Jakarta message service text listener, which functions similarly to an event listener and receives Jakarta message service messages instead of events.

Features of Message-driven Beans:

- They are invoked asynchronously.
- They do not have a state.
- They have a shorter life-expectancy.
- They are capable of recognizing transactions.
- They can access and modify shared information within the database.

Summary

- Session Beans are Enterprise Beans that encapsulate a collection of related business operations or activities and are infused into classes that use them.
- A Session Bean is similar to an interactive session. It can only accommodate one client, similar to an interactive session which has one user.
- Enterprise Beans that provide business tactics to application components and preserve a conversational state with the client are known as Stateful Session Beans.
- A Singleton Session Bean is created once for each application and appears to exist for the entire duration of the application's life cycle.
- An Entity Bean is a business-encoded persistent object that is a component of the Jakarta Enterprise edition.
- A Message-driven Bean is an Enterprise Bean that enables asynchronous message handling in Jakarta EE application forms.

Session: 3

Resource Creation in Jakarta Enterprise Beans

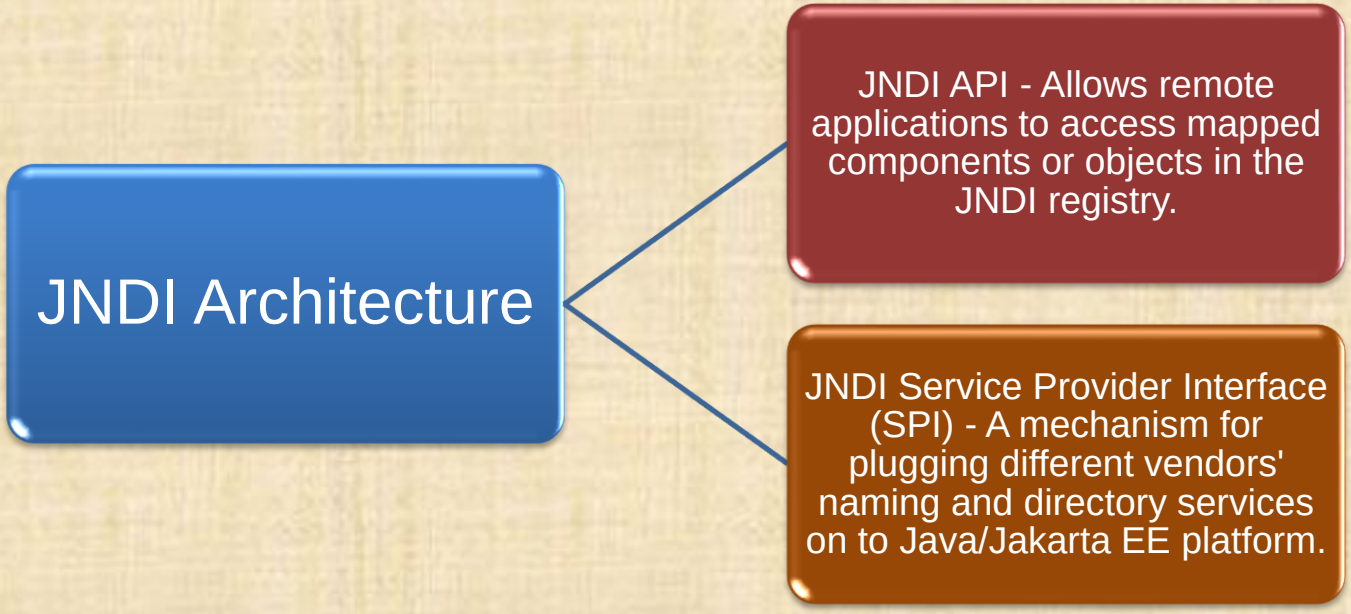
Jakarta Enterprise Beans in Action

Objectives

- Explain Resource and JNDI Naming
- Explain about Data source, Object, and Connection Pools
- Explain the creation of Resources Administratively

JNDI Naming

- Java Naming and Directory Interface (JNDI) is an API that provides naming and directory functionality to application.



JNDI Namespaces

java:global

Format: `java:global[/application name]/module name /enterprise bean name[/interface name ]`

java:module

Format: `java:module/enterprise bean name/[interface name]`

java:app

Format: `java:app[/module name]/enterprise bean name [/interface name]`

Context Object

Code Snippet 1 shows how to retrieve the Context object.

Code Snippet 1:

```
Context ctx = new InitialContext();
```

Methods for binding and unbinding JNDI names with objects are provided by the Context class.

Context Class Methods

Context class provides following methods for binding and unbinding JNDI names with objects:

`bind()`: This method takes two values as arguments , the JNDI name as a string and the object toward which it must be bound.

`unbind()`: This method accepts a string variable as the argument and is utilized to unbind a specific object from a JNDI name.

`lookup()`: This method returns the object within which the lookup is performed.

`rebind()`: Receives two parameters, the JNDI name and the object.

Advantages and Disadvantages of JNDI

Advantages

- Users can use JNDI APIs to write a program without having to know how one's application data is stored in a directory service.
- As long as the resources are accessible from a JNDI directory, you can write code to access data on a single machine or across multiple systems.
- Users can use JNDI to write a program in any programming language that has a JNDI API.

Disadvantages

- The predefined method provided by JNDI can only store definite types of data, and as a result data such as configuration information does not fit into this model.
- No provision is made for transactions.
- There is no built-in security model, though some implementations provide APIs for configuring SSL connections to directory servers.

DataSource, Object, and Connection Pools

DataSource	• Collection of properties to define real world data source
DataSource Object	• Object for physical connection to the data source
Pooled connection	• It is a practice of reusing the connection instead of creating new one.

Connection Pooling

Connection Pooling reuses the connection instead of creating new connections.

Connection Pooling occurs in the background without affecting the running program.

Pooled connection does not have to be explicitly closed

Whenever an application closes a pooled connection, it returns the connection to a pool of reusable connections

Connection pooling allows applications to run much faster because it avoids creating a new physical connection every time when one is requested.

Creating Resources Administratively 1-6

JBoss creates a JNDI `InitialContext` implementation instance for each Web application that runs on it, in a manner similar to that of a Jakarta Enterprise Edition Application Server

The Jakarta EE Standard provides a standard set of elements in the /WEB-INF/application.xml file

Elements to describe resources in Web application:

- `<env-entry>` - An environment entry used to customize the application's behavior.
- `<resource-ref>` - A reference to an object factory for resources characterized in JBoss.

Creating Resources Administratively 2-6

Example for Conversion from Lowercase to Uppercase.

Code Snippet 2: Local Bean Class

```
package demo.jndi;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
@Stateless(mappedName = "Upper")
public class UpperCase implements UpperCaseRemote {
public UpperCase() {   }
          public StringBuffer sayHello() {
               String string1="hello jndi";
                StringBuffer Updates_String=new StringBuffer(string1);

                  for(int i = 0; i < string1.length(); i++) {
                 if(Character.isLowerCase(string1.charAt(i))) {
                          Updates_String.setCharAt(i,
                  Character.toUpperCase(string1.charAt(i)));
                      }
                      else
                  if(Character.isUpperCase(string1.charAt(i)))
                  {
                              Updates_String.setCharAt(i,
                    Character.toLowerCase(string1.charAt(i)));
                      }
                  }
                  System.out.println("String : " + Updates_String);
                              return Updates_String;
          }
}
```

Creating Resources Administratively 3-6

Code Snippet 3: Remote interface

```
package demo.jndi;
import jakarta.ejb.Remote;
@Remote
public interface UpperCaseRemote  {
        public StringBuffer sayHello();
}
```

The set of methods provided by the Session Bean that can be accessed by remote clients is referred to as the remote interface.

Creating Resources Administratively 4-6

Code Snippet 4: Remote Client

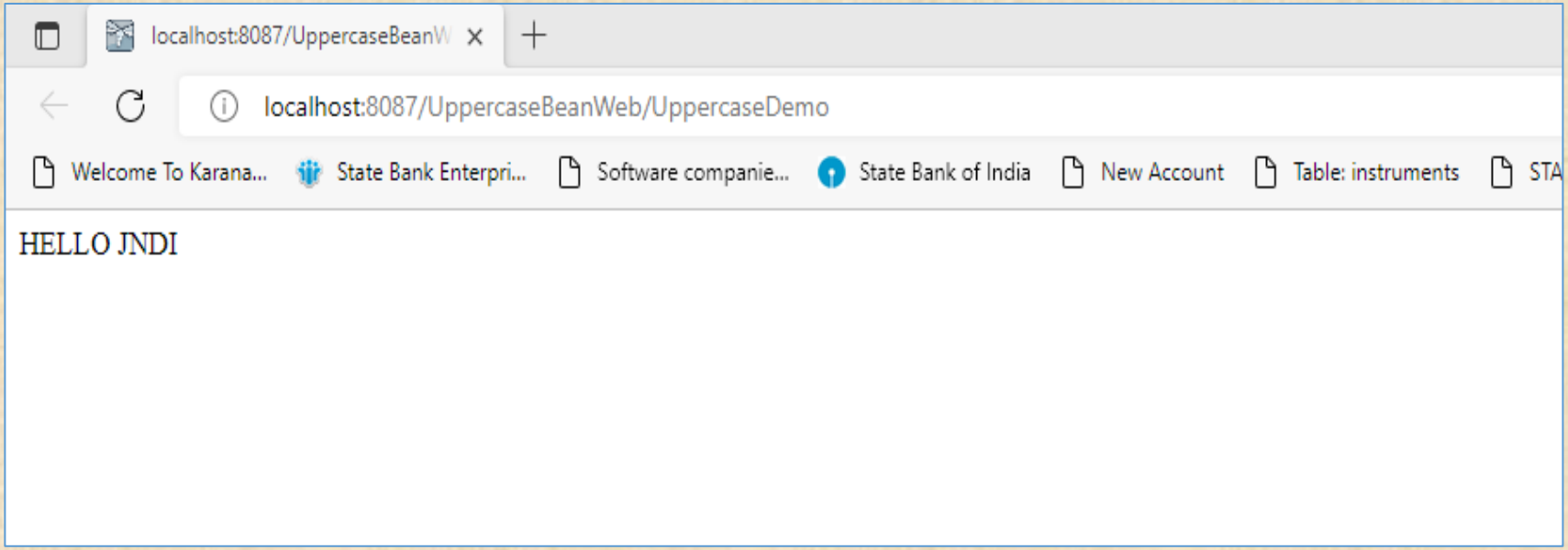
```
package uppercase.jndi;
import java.io.IOException;
import java.io.PrintWriter;
import jakarta.naming.InitialContext;
import jakarta.naming.NamingException;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import demo.jndi.UpperCaseRemote;

@WebServlet("/UppercaseDemo")
public class UppercaseDemo extends HttpServlet {
   private static final long serialVersionUID = 1L;
    public UppercaseDemo() {
        super();
    }
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
```

Creating Resources Administratively 5-6

```
            try {
                InitialContext context = new
                                    InitialContext();
                 UpperCaseRemote remoteBn = (UpperCaseRemote)
                             context.lookup("java:global/Uppercase/
                             UppercaseBean/
                              UpperCase!demo.jndi.UpperCaseRemote");
                 StringBuffer str = remoteBn.sayHello();
                 response.setContentType("text/html");
                 PrintWriter out = response.getWriter();
                 out.print(str);
                 }
                 catch (NamingException e) {
                         e.printStackTrace();
                 }
        }
}
```

Creating Resources Administratively 6-6

Output of the Code Snippets:



Summary

- JNDI is a naming service that is included in Jakarta EE specification. It arranges application elements according to hierarchy and allows for unique identification of application components.
- JNDI API enables remote applications to access mapped components or objects in the JNDI registry.
- JNDI Service Provider Interface (SPI) is a mechanism for plugging different vendors' naming and directory services on to Java EE platform.
- Most applications use a Relational Database to store, organize, and retrieve information. The JDBC API allows Jakarta EE components to connect to Relational Database systems.
- Databases are made accessible via DataSource objects inside the JDBC API.
- JBoss creates a JNDI `InitialContext` implementation instance for each Web application that runs on it, in a manner similar to that of a Jakarta Enterprise Edition Application Server.

Session: 4

Working with Jakarta Enterprise Beans

Jakarta Enterprise Beans in Action

Objectives

- Explain Transaction Processing with Enterprise Beans
- Explain Concurrency Control in Enterprise applications
- Define Event-Driven Programming
- Describe Security in Enterprise applications

Transaction Processing

- Jakarta EE platform provides various underlying services through containers.
- A transaction is a group of operations that are executed as a single unit.
- The information in an Enterprise application is in the form of a database.
- The transaction management mechanism manages transactions performed on the database.
- It is supplied by the Jakarta Transaction API (JTA).

Properties of Transactions

A	C	I	D
Atomicity	**Consistency**	**Isolation**	**Durability**

When all these properties hold good for the reliable processing of a task, the transaction is considered to be valid.

Transaction Demarcation

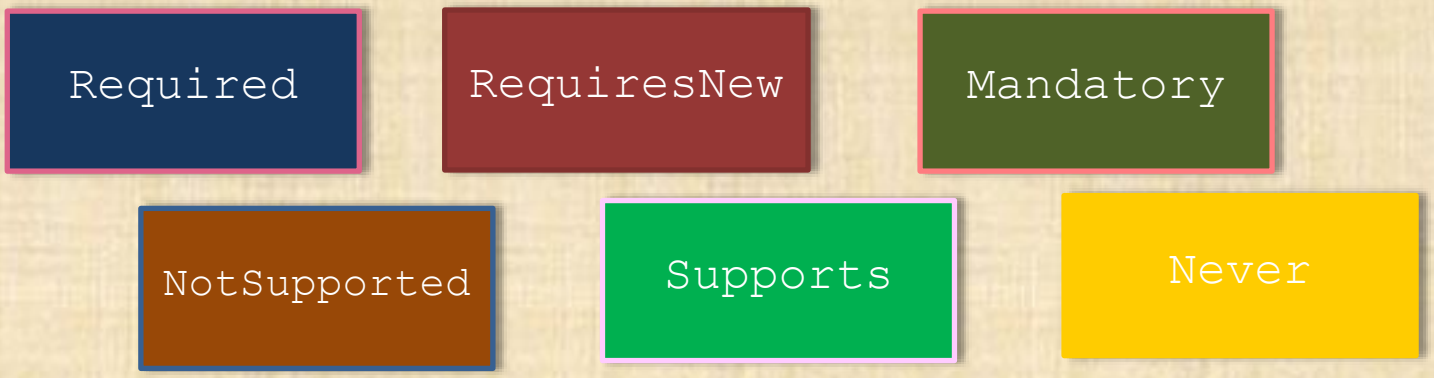
Two methods for transactional demarcation are:

Programmatic Demarcation	Declarative Demarcation
• The developer begins and ends a transaction explicitly by calling the methods for transaction.	• The container controls how and when to use the begin, commit, and abort methods for each transaction.

- The transactions in Jakarta Applications are managed through Jakarta Transaction Service (JTS) and JTA.
- JTA acts as an interface between a transaction manager and the components involved in the distributed transaction system.

Transaction Attributes 1-2

- A transaction attribute controls the limits of that transaction.
- A transaction attribute communicates to the container the concerned transactional behavior of the related method.
- A transaction attribute can have one out of following values:



Transaction Attributes 2-2

Code Snippet shows the example of working of MANDATORY.

A MANDATORY method is assured to be used in each transaction. However, it is the responsibility of the caller to complete the transaction.

Code Snippet 1:

```
@Stateless
public static class MyBean implements MyInterface {
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public String codeBlack(String str) {
return s;
}
public String codeBrown(String str) {
return s;
}
}
```

Concurrency Control

When multiple applications access data, there is concurrent access to the stored data on the database.

Concurrent access to the database is always protected with a technique called isolation.

Database providers provide locking techniques to maintain data integrity.

The two mechanisms used to manage concurrent access of the entities include:

- Optimistic Locking
- Explicit read and write locks (Pessimistic Locking)

Optimistic Locking Vs. Pessimistic Locking

Optimistic Locking	Pessimistic Locking
Checks that no other applications have modified the data.	Acquires the database locks for a long-term on entities.
Acquires no locks for providing concurrency control.	Acquires locks on all the data, before starting the database operations.
Provides a high degree of simultaneous accessing of the data.	Enables data application to be accessed by multiple applications.

Concurrency Management

Singleton Bean is flexible for Concurrency Management as the container can access the Singleton Bean.

A Bean developer can control the Bean concurrency management by applying the annotation `s`.

The probable or enumerated values for the annotation include:

- `ConcurrencyManagementType.CONTAINER`
- `ConcurrencyManagementType.BEAN`

Working of Annotation

Code Snippet 2:

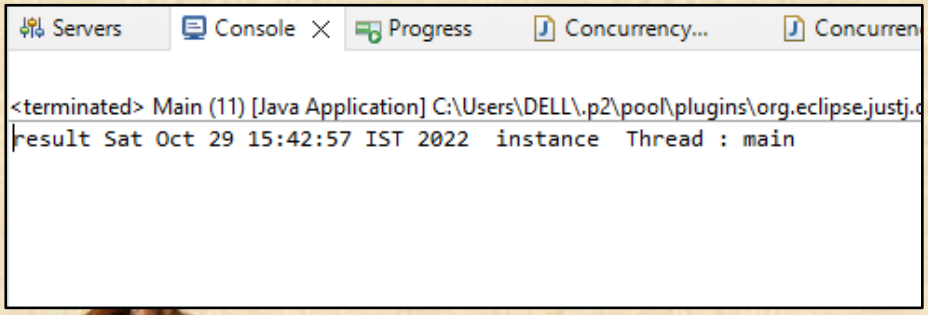
```
package conc.abc;
import java.util.Date;
import jakarta.annotation.PostConstruct;
import jakarta.ejb.ConcurrencyManagement;
import jakarta.ejb.ConcurrencyManagementType;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Singleton;
@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CO
NTAINER)
public class ConcurrencyDemo1 implements
ConcurrencyDemo1Remote {
private String property;
public void update(String property){
this.property = this.property + " " + property;
}
@PostConstruct
public void init() {
property = new Date().toString();
}
public ConcurrencyDemo1() {}
}
```

Remote Interface Creation

Code Snippet 3:

```
package conc.abc;
import jakarta.ejb.Remote;
@Remote
public interface
ConcurrencyDemo1Remote {
public void init();
}
```

Output of Application for Remote Access:



Code Snippet 4:

```
import java.util.Date;
import conc.abc.ConcurrencyDemo1Remote;
public class Main {
public static ConcurrencyDemo1Remote
ConcurrencyDemo1;
public static void main(String[] args) {
ConcurrencyDemo1Remote ConcurrencyDemo1=
new
conc.abc.ConcurrencyDemo1();
System.out.println("result " + new
Date().toString() + "instance Thread :
"+
Thread.currentThread().getName());

}
public Main() {
    super();
  }
}
```

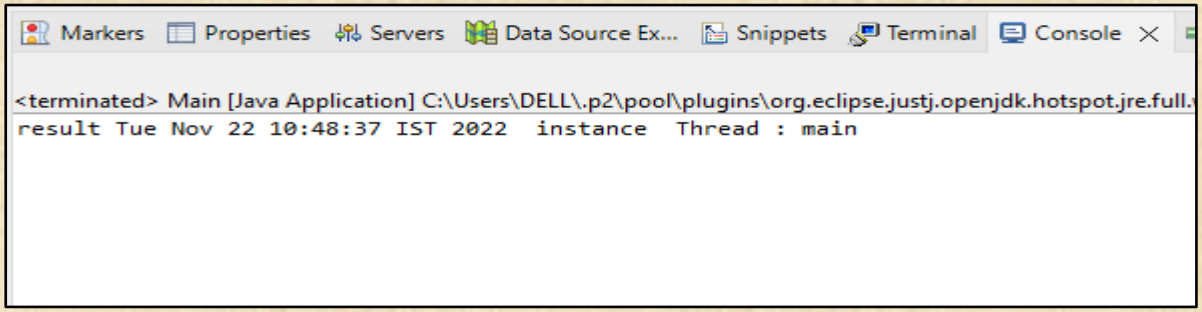
Concurrency Control Using Timeout

Code Snippet 5 shows a sample example using Timeout.

Code Snippet 5:

```
package ejb.demo.conc;

import jakarta.ejb.AccessTimeout;

import jakarta.ejb.Stateful;

@Stateful

public class BeanWithTO{

@AccessTimeout(value=4000,

unit=java.util.concurrent.TimeUnit.MILLIS
ECONDS)

public void find(String id){

//...your code

}

}
```

Execution of Application



Event-driven Programming

- Event-driven programming is a programming paradigm where the program's flow depends upon events.
- Jakarta Messaging Service (JMS) messages are used to invoke MDBs.
- Dependency and Contexts Injection is a feature of Jakarta EE 6 onwards.

- The Session Enterprise Beans can have annotations such as `@AroundConstruct`, `@LostConsgtruct`, `@PreDestroy`, `@PostActivate`, and `@PrePassivate`.
- Developers can define their own Events and Observers and fire them from any managed Bean inside the application.

Security in Enterprise Applications

- Container is responsible to provide security services for the components deployed in it.
- In Jakarta EE, the security policy is implemented through the code written for the application by means of annotations and deployment descriptors.
- Enterprise application security can be implemented at three different levels along with the code for the application.

Application Layer Security

- All the EJB components are implemented in the container on the application layer.
- The security for the application layer is implemented by the application container. Firewalls can be used inside the application layer level.
- Annotations can specify security information pertaining only to the class in which it is invoked.
- Deployment descriptors are XML files that are used to specify security information for the application.
- Enterprise bean applications use `ejb-jar.xml` as a deployment descriptor and Web applications use `web.xml`.

Transport Layer Security

- The transport layer uses HyperText Transfer Protocol (HTTP) using SSL or Secure Sockets Layer.
- The transport layer security uses cryptographic techniques for security.
- Transport layer security is unaware of message contents that are being transmitted.
- The security information is bundled within the Simple Object Access Protocol (SOAP) message.
- The security information travels to the destination as a part of the message.
- The encrypted SOAP message is only decrypted by the receiver.

Message Layer Security

- Message layer security is also called under the name end-to-end security.
- The encrypted SOAP message is only decrypted by the receiver.
- Unlike transport layer security, message layer security can be selectively applied on a portion of the message.
- Message layer security may be implemented independent of application environment.

Summary

- A Jakarta container provides support services such as transaction management, security, and so on.
- An Enterprise Application stores critical information for business operations in the databases.
- The database management system lists four properties known as the ACID attributes in order to ensure data integrity.
- ACID properties attributes stand for Atomicity, Consistency, Integrity, and Durability.
- Transaction isolation strategies ensure that concurrent access to the database is always secured..
- The most widely employed method in the applications is optimistic locking.
- Persistence providers use pessimistic locking when data integrity is crucial for certain applications.
- In Jakarta EE, the security policy is implemented within the application code through annotations and deployment descriptors.

Session: 5

Facelets in Jakarta Enterprise Beans

Jakarta Enterprise Beans in Action

Objectives

- Define Facelets
- Explain lifecycle of Facelets
- Explain how to create a simple Facelet Application
- Define Composite Components
- Define Web Resources

Introduction to Facelets 1-2

- JSF or Java Server Faces is one of the UI frameworks for Java applications. It is a server-side framework.
- Features of Jakarta Server Faces:

Easy to move application data to and from the user interface.	Client-side events and server-side application model can be connected easily.
Facilitates the creation and reuse of customised UI components.	Facilitates the creation of a UI from a collection of reusable UI components.

Introduction to Facelets 2-2

- Facelets is a powerful, but lightweight page declaration language that is used to create component trees and build JavaServer Faces views.
- JSF can handle components as stateful objects on the server and map HTTP requests to component-specific event handling.
- Web applications can be created using JSF technology.
- JSF supports the XML namespace and creates components in the HTML-style Web page.

Advantages of Facelets

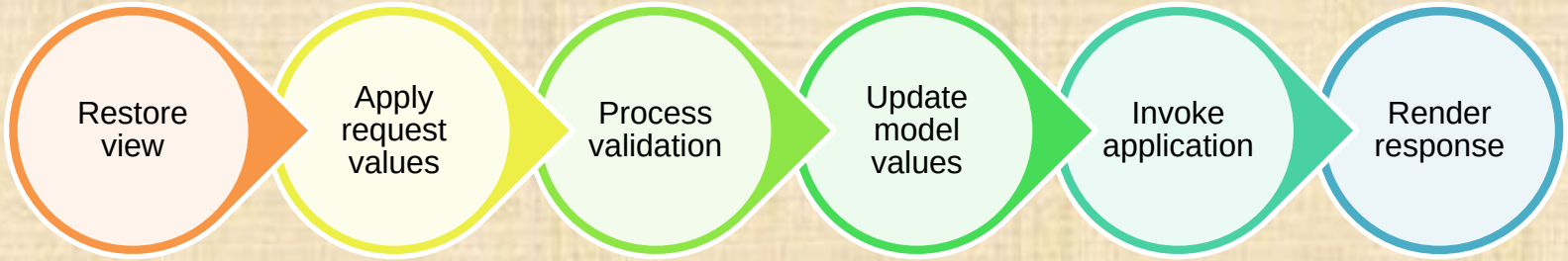
Promotes reusability through Composite components and templates.	Promotes abstract aggregation and code reusability through Composite components.	Provides faster compilation time, security, and faster rendering of views.
Enables functional extensibility through Components customization.	Builds on JavaServer Faces and Jakarta Server Faces MVC Web framework.	Reduces the time and labor required for development and deployment.

Facelets Library Tag List

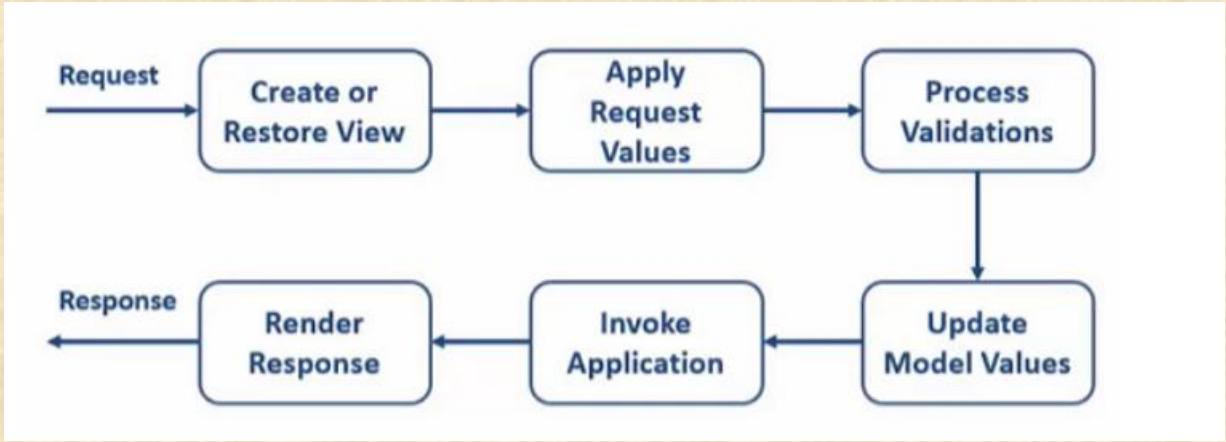
ui:component	ui:decorate	ui:composition
ui:insert	ui:remove	ui:repeat
ui:param	ui:include	ui:fragment

Lifecycle of Facelets

- Execute and Render are two main phases in the lifecycle of Facelets.
- Steps under the Execute phase:



- Lifecycle of JSF and the request processing via JSF.



Creating Simple Facelet Application 1-3

Code Snippet 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/ja
vaee
http://xmlns.jcp.org/xml/ns/javaee/web-
app_4_0.xsd"
id="WebApp_ID" version="4.0">
<display-name>HelloWorldFacelet</display-name>
<welcome-file-list>
<welcome-file>index.html</welcome-file>
<welcome-file>index.jsp</welcome-file>
<welcome-file>index.htm</welcome-file>
<welcome-file>default.html</welcome-file>
<welcome-file>default.jsp</welcome-file>
<welcome-file>default.htm</welcome-file>
</welcome-file-list>
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-
class>jakarta.faces.webapp.FacesServlet</s
ervlet-
class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
<context-param>
<description>State saving method:'client'
or 'server'
(=default). See JSF Specification
2.5.2</description>
```

Creating Simple Facelet Application 2-3

```
<param-
name>javax.faces.STATE_SAVING_METHOD</param
-name>
<param-value>client</param-value>
</context-param>
<context-param>
<param-
name>javax.servlet.jsp.jstl.fmt.localizatio
nContext</param-name>
<param-value>resources.application</param-
value>
</context-param>
<listener>
<listener-
class>com.sun.faces.config.ConfigureListene
r</listener-class>
</listener>
</web-app>
```

Code Snippet 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<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
http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd"
bean-discovery-mode="all"
version="2.0">
</beans>
```

Creating Simple Facelet Application 3-3

Code Snippet 3:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://xmlns.jcp.org/jsf/facelet
s"
xmlns:h="http://xmlns.jcp.org/jsf/html"
xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
<title> Hello</title>
</h:head>
<h:body>
<p>Hello world</p>
<h:outputText value="#{helloWorld.str}"/>
</h:body>
</html>
```

Code Snippet 4:

```
package jsf.hello.co;
import java.io.Serializable;
import jakarta.enterprise.context.SessionScoped;
import jakarta.inject.Named;
@Named
@SessionScoped
public class HelloWorld implements Serializable{
private static final long serialVersionUID = 1L;
private String str = "Hello, we have created
first Jakarta Server Faces code";
public String getStr() {
            return str;
}
public void setStr(String str) {
            this.str = str;
}
}
```

Output

Hello world

Hello, we have created first Jakarta Server Faces code

Composite Components

- Composite components are a part of Jakarta Server Faces along with Facelets.
- Composite Components promote uniformity in the application, code re-use, and maintain application code at large.
- With collection of markup tags, and other components, developer can encapsulate the functionality.
- Composite component library tags are:

```
composite:implementation
composite:interface
composite:actionSource
composite:editableValueHolder
composite:attribute
composite:insertChildren
composite:valueHolder
```

Advantages of Composite Components Facelets

- Composite Components are used especially to create custom UI components.
- A proper and well-defined event handling.
- Faster and accessible Web applications.
- Well defined data attributes.
- A facilitated response and request to the server.

Disadvantages of Composite Components

- CC does not provide proper specifications for complex component Facelets.
- Composite components are severely constrained. Components should be written as classic, Java-centric JSF components.
- The performance of composite components lags behind standard JSF components.

Web Resources

- Software components such as CSS files, Script files, Images, and Jar files are considered Web resources.
- For a developer, a minimum resource name in the directory path such as Stylesheet, Script, or Images.
- Resources are placed mainly in a folder, under the root of the Web application in a folder with ‘WEB-INF’.
- In the JSF Web application, all Web resources are treated to exist in a library location resource directory.
- A composite component or a template is stored in the resources directory and can be used to create a resource instance.

Summary

- One of the User Interface (UI) frameworks is JavaServer Faces (JSF). In Jakarta EE, it is called Jakarta Server Faces.
- XHTML pages are typically used to create Facelets views. The XHTML Transitional Document Type Definition (DTD).
- A lifecycle of the Jakarta Server Faces application starts when a client sends an HTTP request for a page and ends when the server returns the page as an HTML response.
- The Jakarta Server Faces and Facelets specification defines the JSF as well Facelets lifecycle.
- Facelets is a JSF extension to create Web applications having uniformity across Web applications created as XHTML pages.
- Facelets promotes code reuse by using its templating feature.
- Facelets acts as a Web application UI container for Jakarta Server Faces UI components and elements.

Session: 6

Jakarta Local and Remote Clients

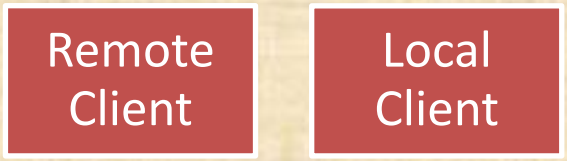
Jakarta Enterprise Beans in Action

Objectives

- Define Jakarta Local and Remote Client
- Explain when to use Local Client and Remote Client
- Explain the method of choosing between Local and Remote Client
- Outline client view on Object lifecycle
- Describe object identity

Overview of Jakarta Local and Remote Client 1-4

- An entity bean is a client-side component that represents an object-oriented view of some entities stored in storage devices, such as a database, or entities executed by an existing enterprise application.
- A client of entity bean can be a Local Client or a Remote Client.



- An entity object stays in a container from the time it is created until it is destroyed.
- The container transparently offers protection, concurrency, transactions, perseverance, as well as other services to the entity objects that reside in the container.

Overview of Jakarta Local and Remote Client 2-4

- An entity object can be accessed by multiple clients at the same time.
- Transactions are used by the container in which the entity bean is deployed to properly synchronize access to the state of the entity object.
- Each entity object does have an identity that, in most cases, survives a crash and restarts the container in which it was created.
- The container implements object identity with the help of the enterprise bean class.
- A container can hold multiple enterprise beans.
- The container provides a class that implements home interface of an entity bean for each entity bean deployed in it.

Overview of Jakarta Local and Remote Client 3-4

- A client can obtain the entity home interface of entity bean via dependency injection or by using JNDI (**JNDI** stands for Java Naming and Directory **Interface).**
- It is the responsibility of the container to make the entity bean's home interface available in the JNDI namespace.

Overview of Jakarta Local and Remote Client 4-4

- There can be different types of clients to a Session bean. These clients can access the Session bean either remotely or locally.

Remote Client

It can be standalone Jakarta application, A web service, or other session bean executing as a part of enterprise application in different JVM.

Local Client

It can be components residing within the same enterprise application. It can either be another session bean or a Web component servlet.

Local Client

- A Local Client can access the enterprise bean through any one of following methods:

No-Interface view	Business Interface
Through no-interface view, the Local Client can access the enterprise bean through the public methods defined in the bean class.	The business interface can be annotated with `@ Local`. The interface can be implemented through a different class, where the interface class names is provided as a property of the @ Local annotation as, @ Local (interface.class)

Understanding Jakarta Local Client with an Example 1- 2

To print prime numbers from 1 to 100

Steps to create the application are as follows:

- **Step 1**: To create a Project Go to New->Project->EJB project as shown in Figure 6.1.

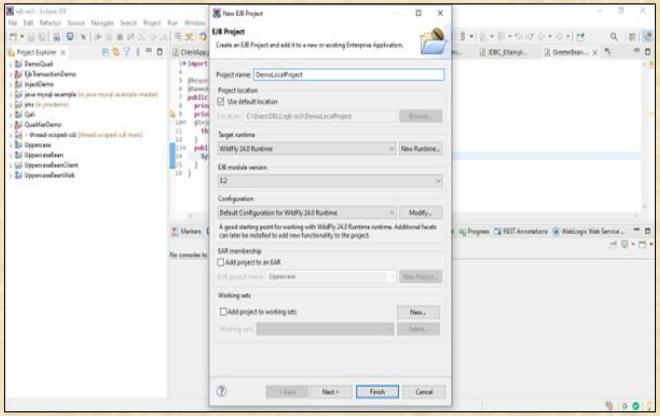


Figure 6.1: Project Creation

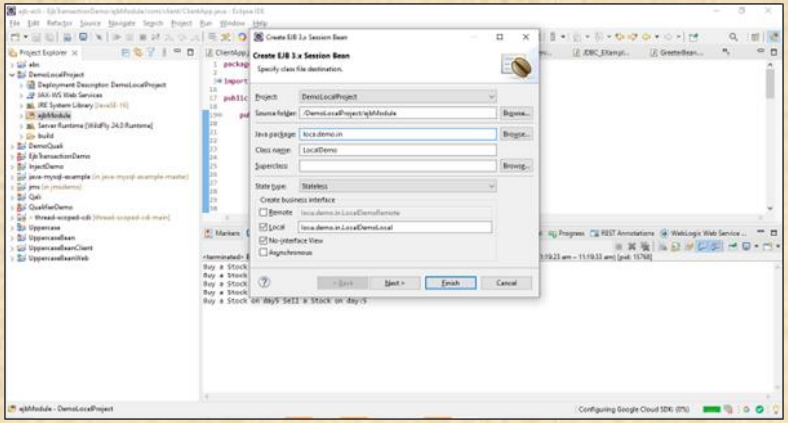


Figure 6.2: Adding Session Bean

- **Step 2**: Add Session bean with local interface as shown in Figure 6.2.

Understanding Jakarta Local Client with an Example 2-3

Code Snippet for prime numbers from 1 to 100.

Code Snippet 1:

```
package loca.demo.in;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
@Stateless
@LocalBean
public class LocalDemo implements
LocalDemoLocal
{
public void Prime() {
      int i =0;
      int Number =0;
     String primeNumbers = "";
     for (i = 1; i <= 100;i++)
      {
         int Prime_counter=0;
         for(Number =i; Number>=1;
            Number--) {
            if(i%Number==0) {
Prime_counter = Prime_counter + 1;
    }
  }
```

```
  if (Prime_counter ==2)
   {
     primeNumbers =
primeNumbers + i + " ";
   }
 }
  System.out.println("Prime
numbers from 1 to
100 are :");
 System.out.println(
primeNumbers);
   }
}
}
```

Jakarta Local Client with an Example 3-3

- Local interface for the `LocalDemo` class.

Code Snippet 2:

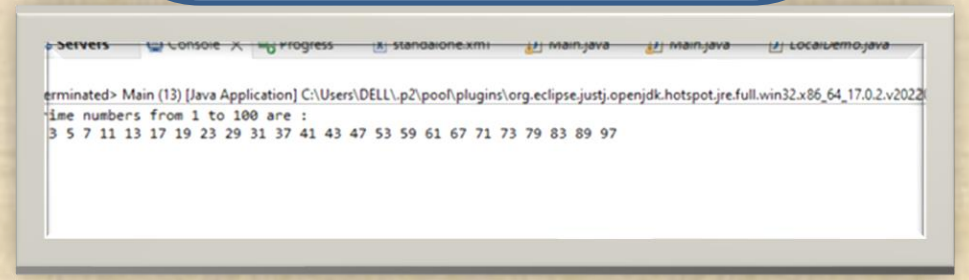
```
package loca.demo.in;
import jakarta.ejb.Local;
@Local
public interface LocalDemoLocal {
public void Prime();
}
```

- `Client` Class

Code Snippet 3:

```
import loca.demo.in.LocalDemo
Local;
public class Main {
public static void
main(String[] args) {
LocalDemoLocal StateLess=
new
loca.demo.in.LocalDemo();
StateLess.Prime();
}
}
```

- Figure demonstrates the execution of an application that uses Local Client.



Output of Application Using Local Client

When to Use Local Client?

- When the bean and the client are in the same JVM, a Jakarta Enterprise Beans local business interface declares the methods that are obtainable.
- A remote and local interface can exist in an EJB component.
- If an EJB component has a local interface, Local Clients running on the same instance of the application server can utilize it instead of the remote interface.
- If a bean is only to be utilized by Local Clients, only the local interface should be provided.

Remote Clients

- A Remote Client can run in a different application, different machine, or a different JVM.
- A Remote Client can be a bean component, a Web component, or an application client.
- Remote Clients can access the enterprise bean through a remote interface.
- A Remote Client to an EJB may not be located within the same JVM to access the methods of the bean.
- The Remote Client can be another enterprise bean residing in the same or different location, or it can also be a Web application, an Applet, or a Jakarta console application.
- The EJB client API was introduced in WildFly 26 to manage remote EJB invocations.

Understanding Jakarta Remote Client with an Example 1-4

In WildFly, users could either use the EJB client API specific to WildFly or use JNDI to lookup a proxy for one's bean and invoke on that returned proxy.

An example for printing a string in a Reverse order.

- **Step 1**: To create a Project, click File -> New ->EJB project.
- **Step 2**: Add Session bean with Remote interface as shown in Figure.

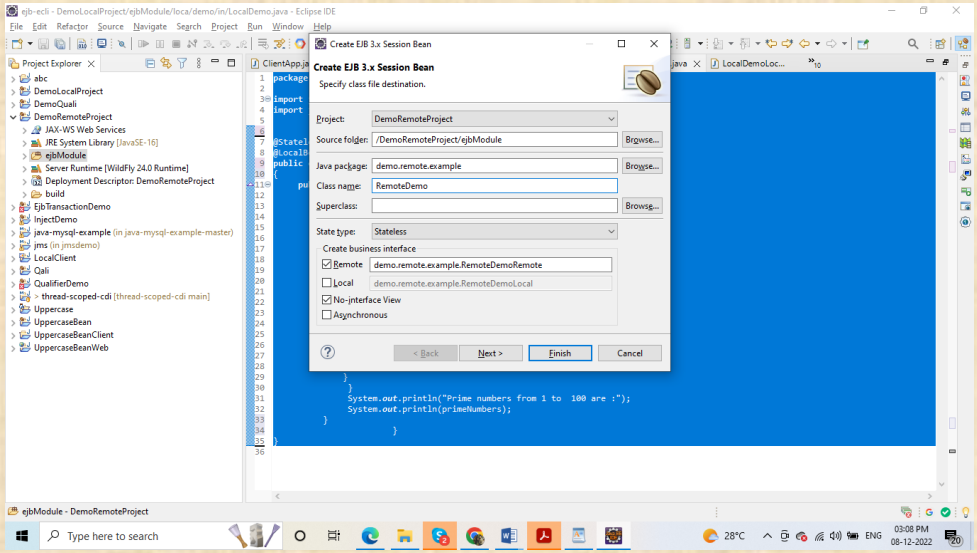


Figure: Adding Session Bean

Understanding Jakarta Remote Client with an Example 2-4

- Code Snippet 4 shows the Remote Client for printing a string in a Reverse order.
- Code Snippet 5 shows the Remote interface for the `RemoteDemo` class.

Code Snippet 5:

```
package demo.remote.example;
import jakarta.ejb.Remote;
@Remote
public
interface RemoteDemoRemote {
public void Reverse();
}
```

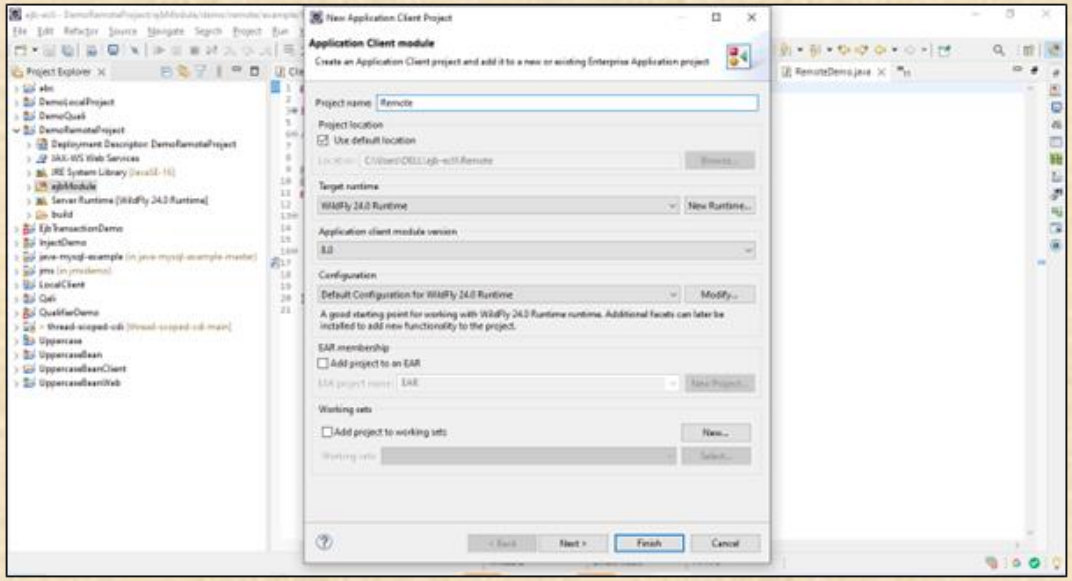
Code Snippet 4:

```
package demo.remote.example;

import java.util.Scanner;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
@Stateless
@LocalBean
public class RemoteDemo implements
RemoteDemoRemote {

public void Reverse() {
String Og_String, Rev_String = "";
    Scanner in = new Scanner(System.in);
    System.out.println("Enter a string to
reverse");
    Og_String = in.nextLine();
    int length = Og_String.length();
    for (int i = length - 1 ; i >= 0 ; i--)
    Rev_String = Rev_String +
Og_String.charAt(i);
    System.out.println("Reverse of the
string: " + Rev_String);
    }
}
```

Understanding Jakarta Remote Client with an Example 3-4

❑ **Step to create the client:**

Click File → New, select Application Client Project, and specify project name as shown in Figure.



Application Client

Understanding Jakarta Remote Client with an Example 4-4

- Code Snippet 6 shows how a Remote client can access the `RemoteDemo` application.

- Figure demonstrates the execution of a Remote Client application.

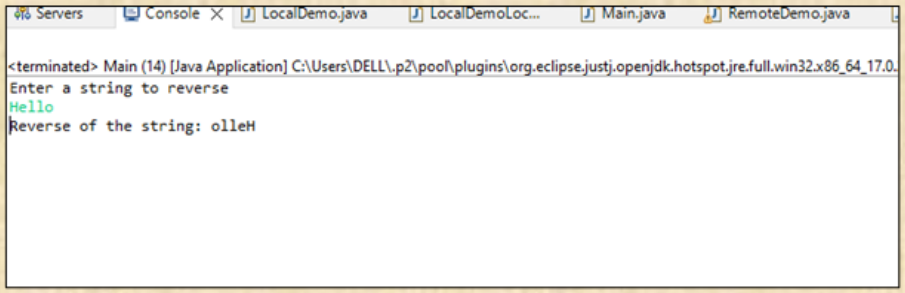


Figure : Output of Remote Client

Code Snippet 6:

```
import demo.remote.example.Remote
DemoRemote;
public class Main {
public static void
main(String[] args) {
RemoteDemoRemote StateLess= new
demo.remote.example.RemoteDemo();
StateLess.Reverse();
}
}
```

When to Use Remote Client?

- A Remote Client of an enterprise bean shares following characteristics:

Remote Clients cannot access an enterprise bean via a no-interface view.

When the bean is accessed from a different environment, remote business interfaces are required.

when the client is located on another Jakarta Virtual Machine, use the Remote Client view.

- Remote Client is independent of the enterprise bean it is accessing and can run on a different machine and JVM
- Remote Client could be a Web component, a client application, or perhaps another enterprise bean.
- The location of the enterprise bean is a transparent to a Remote client.
- A business interface must be implemented by the enterprise bean.

Choosing Between Local and Remote Client

- When deciding whether to employ local or remote access for an entity bean, following factors should be considered:

While using Enterprise Beans 2.1 or earlier versions remote home and remote component interfaces, relatively narrow remote types necessitates usage of `javax.rmi.PortableRemoteObject.narrow` instead of Java language casts.

Remote calls could result in error situations as a result of interaction, resource utilization on some other servers, and other factors that are not anticipated in local calls. While using Enterprise Beans 2.1 and earlier versions remote home and remote component interfaces, the client must expressly create handlers for `the java.rmi.RemoteException`.

Client View of Session's Object Lifecycle

A session object, which does not exist cannot be referred. Invocations of a session object that does not exist fail with `java.rmi.NoSuchObjectException` is thrown/encountered.

A client can perform following actions with access to a session object:

- Call business methods defined in the remote interface of the session object.
- Retrieve a reference to the home interface of the session object.
- Within the scope of the client, pass the reference as a parameter or return value.
- Retrieve the session object's handle.
- Delete the session object. When the session object's lifetime expires, a container may also remove it automatically.

Session Object Identity

- Entity objects expose their identity as a primary key and session objects hide their identity.
- The `EJBObject.getPrimaryKey()` and `EJBHome.remove (Object primaryKey)` methods result in a `java.rmi.RemoteException` if called on a session bean.
- If the `EJBMetaData.getPrimaryKeyClass()` method is invoked on a `EJBMetaData` object for a session bean, the method throws the `java.lang.RuntimeException.`
- Session objects are meant to be private resources that only the client that formed them can access.
- Unless called on a session bean, the `EJBObject.getPrimaryKey()` and `EJBHome.remove(Object primaryKey)` methods throw a `java.rmi.RemoteException.`
- When the `EJBMetaData.getPrimaryKeyClass()` method is called on an `EJBMetaData` object for a session bean, it throws the `java.lang.Exception` exception during runtime.

Summary

- An entity bean is a client-side component that represents an object-oriented view of some entities stored in storage devices.
- Session bean clients can access the Session bean either remotely or locally.
- An entity bean's client can be a Local Client or a Remote Client.
- A local client runs in the same application that enterprise bean is accessing.
- Local Clients that access the enterprise bean's no-interface view have access to the public methods of the enterprise bean implementation class.
- A Remote Client can run in a different application, different machine, or a different JVM.

Session: 7

Jakarta Messaging Services

Jakarta Enterprise Beans in Action

Objectives

- Describe Jakarta Messaging
- Explain the goals of Jakarta Messaging, Jakarta Messaging Domains, and the importance of JMS
- Explain Jakarta Messaging Architecture, Jakarta Messaging Application, Jakarta Messaging APIs, classic API interfaces, and simplified API Interfaces
- Elaborate the process of developing a Jakarta Messaging Application

Introduction

Jakarta Messaging

- Jakarta Messaging was initially created to serve as a standard Java API for the existing messaging products.
- Jakarta Messaging unifies the use of messaging products by Java client applications and Java middle-tier services.
- Jakarta Messaging describes a few messaging semantics as well as a set of Java interfaces.
- Since messaging is a peer-to-peer technology, Jakarta Messaging users are referred to as clients.
- A Jakarta Messaging application consists of a collection of application-defined messages and clients that transfer them.

Objectives of Jakarta Messaging

- Provide messaging functionality to Java applications in order to implement advanced enterprise applications.
- Describe a similar set of messaging ideas and facilities.
- Reduce the number of concepts that a Java language programmer must learn in order to use enterprise messaging products.
- Achieve maximum Java messaging application adaptability between messaging products.

Jakarta Messaging Domain

Messaging requires the developer to decide on the messaging style or domain to be adopted.

There are two messaging models supported by the JMS API:

- Publish-Subscribe model
- Point-to-Point model

The message consumption is by default asynchronous in both the messaging model as the message consumers do not actively wait on the destination objects.

Publish-Subscribe Model

Different types of subscription:

Sharable subscription	Non-Sharable subscription	Durable subscription	Non-Durable subscription
• Implies that the message can be received by a set of consumers who have agreed to share the subscription.	• Implies that only one subscriber can receive a message through subscription.	• Messages are retained by the topic for a certain time period according to the configuration of the application.	• The messages are stored on the topic only when there is some client to receive the message and after all the subscribed clients receive the message it is removed from the topic.

Point-to-Point Model

- In Point-to-point messaging model, the communication is between a pair of components. The destination in this case is a queue object.
- User can only have a single consumer for each message. A consumer can grab the message in the queue, but a given message is consumed only once by a consumer.
- Multiple producers can send messages to a queue, but only a single consumer can consume a message.
- Messages are distributed on First In First Out (FIFO) basis.
- The message consumers can also configure a message listener to listen to the message queue and invoke appropriate methods when a message is received in the queue.

Importance of JMS

- The API allows the application components to set up a message listener for processing the JMS messages.
- Various components such as application clients, enterprise beans, and Web components can send messages and asynchronously receive JMS messages.
- Containers can maintain a pool of message driven beans to enable concurrent processing of messages.
- Java EE also provides Message-driven Beans which are enterprise components. These Message-driven beans are used to process JMS messages.
- JMS messages can also be part of a Java transaction, which means that a Java transaction can send or receive a message.

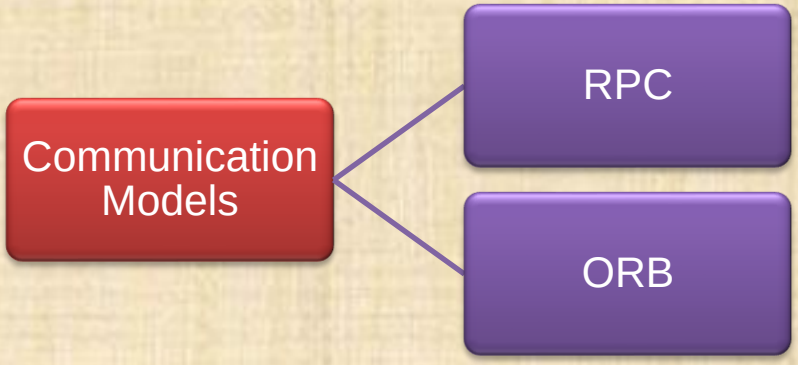
Jakarta Messaging Architecture 1-2

The Java Remote Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) is an extension of RMI. The RMI-IIOP is the protocol of the enterprise Java Beans developed in the Jakarta enterprise application.

Benefits of Messaging:

- **Non-blocking request processing** or **Asynchronous processing**
- **Decoupling**
- **Reliability**
- **Support for Multiple Producer and Consumer**

Jakarta Messaging Architecture 2-2



Remote Process Call

- This type of communication is also referred as synchronous messaging.
- The same mechanism is adopted by Jakarta to allow communication between two objects using RMI.

Object Request broker

- It is also known as ORB-based middleware which is designed to allow the communication between two systems which may be running on different platforms.

Jakarta Messaging Services API Classification

The JMS API provides the functionality of creating, sending, receiving, and reading messages among the application components. It also defines a set of interfaces which can be used in applications to implement the messaging function.

The SPI is used as a plug-in for the Message-Oriented Middleware implementation on the server. It is a JMS provider which talks to server specific Message-Oriented Middleware implementation.

Components of Jakarta Messaging Application

- Jakarta Messaging Clients
- Non-Jakarta Messaging Clients
- Messages
- Administered Objects
- Jakarta Messaging Provider

Jakarta Messaging APIs

Each API offers a distinct set of interfaces for attempting to connect to, and sending and receiving messages from, a Jakarta Messaging provider.

Shared Primary Interfaces	Objects of JMS Application
Queue	Administered objects
`Message`, `BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage`, and `TextMessage`.	Connections
Topic	`JMSContext`
Destination	JMS `MessageProducers`
	JMS `MessageConsumers`
	Messages

Classic API Interfaces

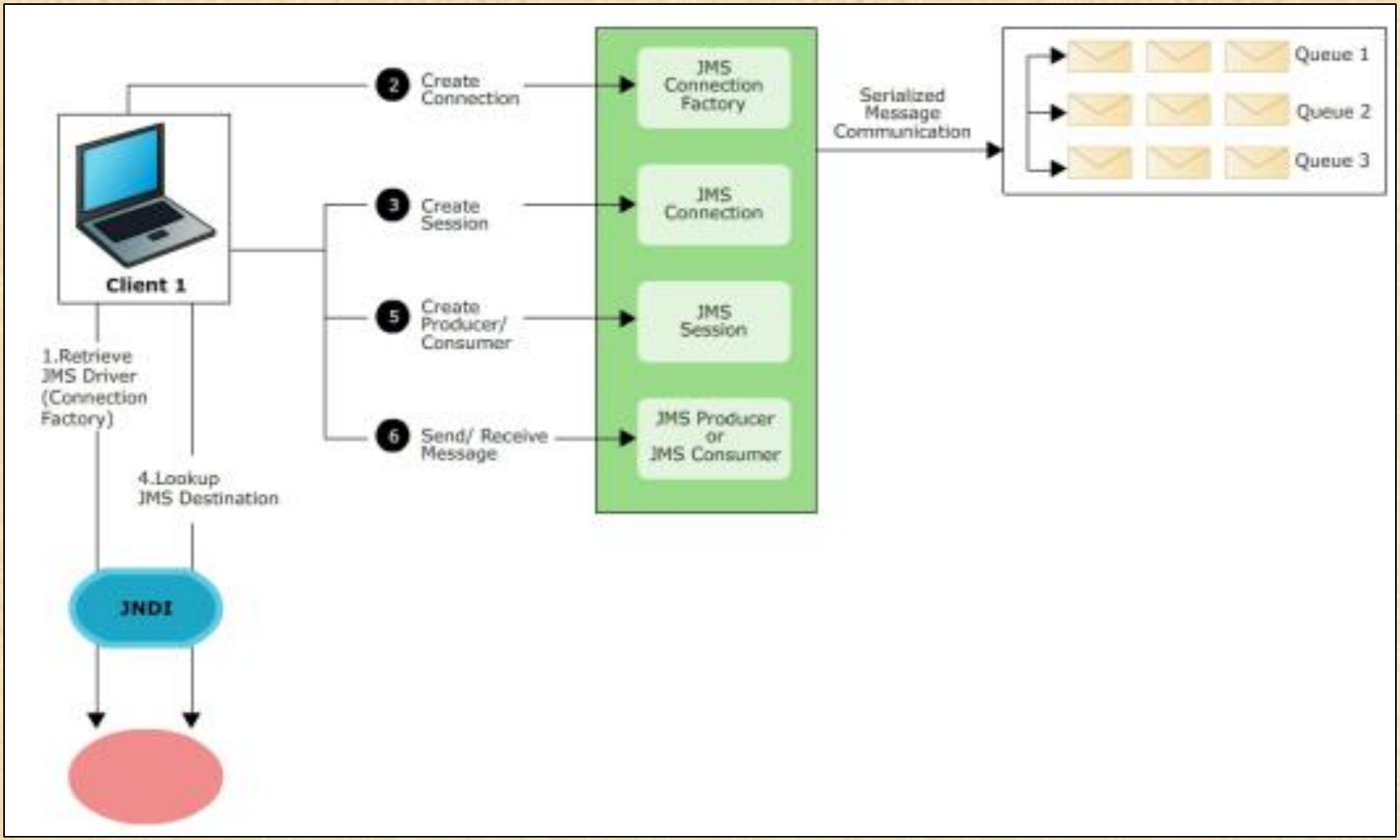
Primary interfaces provided by the classic API are:

- Connection
- ConnectionFactory
- Session
- MessageConsumer
- MessageProducer

Simplified API Interface

`JMSContext`	• A live connection to a Jakarta Messaging provider, as well as a single-threaded context for sending and receiving messages.
`ConnectionFactory`	• A managed object that a client uses to create a Connection. The classic API also makes use of this interface.
`JMSConsumer`	• A `JMSContext`-created object used to receive messages sent to a queue or topic.
`JMSProducer`	• A `JMSContext`-created object used to send messages to a queue or topic.

JMS API Programming Model



Differences between Normal and Conventional Jakarta Messaging Client

Normal Jakarta Messaging Client	Conventional Jakarta Messaging Client
Uses classic API.	Uses Streamlined API.
Uses JNDI to locate one or more Destination objects.	Uses JNDI to locate one or more Destination objects.
Uses JNDI to locate `ConnectionFactory`, Creating a Jakarta Messaging Connection object with message delivery disabled using `ConnectionFactory.`	Uses JNDI to locate `ConnectionFactory` object.
Creates the `MessageProducer` and `MessageConsumer` objects required by using session and destinations.	Generates the `JMSProducer` and `JMSConsumer` objects required using the JMSContext.
Creates one or more Jakarta Messaging Session objects using the Connection.	Generates a `JMSContext` object, use the `ConnectionFactory.`
Tells the Connection to begin message delivery.	Message delivery begins automatically.

Summary

- A Jakarta Messaging application consists of a collection of application-defined messages and clients that transfer them.
- The messaging domain defines the pattern of communication between the two components of the applications. The pattern of communication decides the technology used for exchanging information between the applications.
- In Point-to-point messaging model, the communication is between a pair of components. The destination in this case is a queue object.
- JMS API is an integral part of Jakarta Enterprise Edition. It was first part of J2EE 1.2 as JMS 1.0. Since then, it has evolved to JMS 2.0 which is the most recent version in Jakarta EE.
- JMS clients are those application components that use the services provided by the JMS provider and exchange messages among themselves.
- The simplified API supports the very same messaging functionality as the classic API.
- Jakarta Messaging lacks features for controlling or configuring message integrity or privacy.
- A client can utilize the `JMSCorrelationID` header field to connect two messages. A common application is to connect a response message to its request message.

Session: 8

Understanding Jakarta Connectors Architecture

Jakarta Enterprise Beans in Action

Objectives

- Explain about Jakarta Connectors in detail
- Elaborate on JDBC connectors
- Outline the architecture of Jakarta Connectors
- Explain Resource Managers and connections
- Identify and describe application components

Jakarta Connectors

Connects Jakarta components and EIS Systems.

- Connector makes it easier to integrate various EIS.

Implementation is required only once.

- An implementation is transferable.

EIS vendors can offer a common resource adapter which enables communication between the enterprise application, the application server, and the EIS.

- An application server can accept multiple resource adapters.

Jakarta Connectors Architecture

A standard set of scheme contractual agreements between an application server and EIS, as defined by the connector architecture.

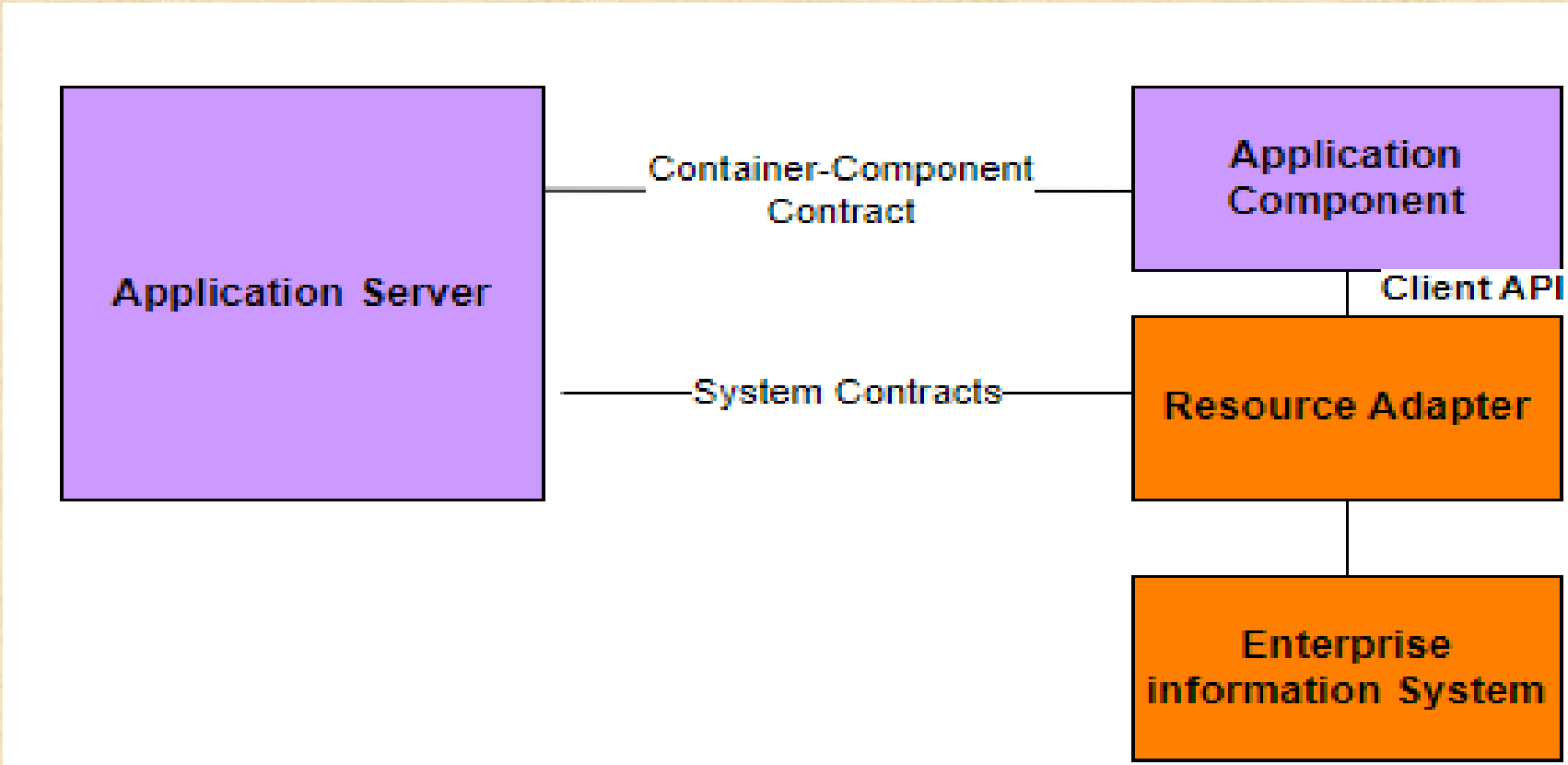
A client API for interacting with multiple EISs defined by a Common Client Interface (CCI).

A resource adapter deployment and packaging protocol that is consistent.

JDBC and Jakarta Connectors

- JDBC API offers a mechanism for transmitting SQL statements to databases and processing the tabular data returned by the databases.
- Jakarta connector Architecture is a standard design for integrating Jakarta application forms with non-relational EISs.
- Each one of the EISs offers a native function call API.
- CCI was designed to offer an EIS-independent API for coding these EIS function calls.
- Architecture also defines a standard Service Provider Interface (SPI).

Jakarta Connectors Architecture

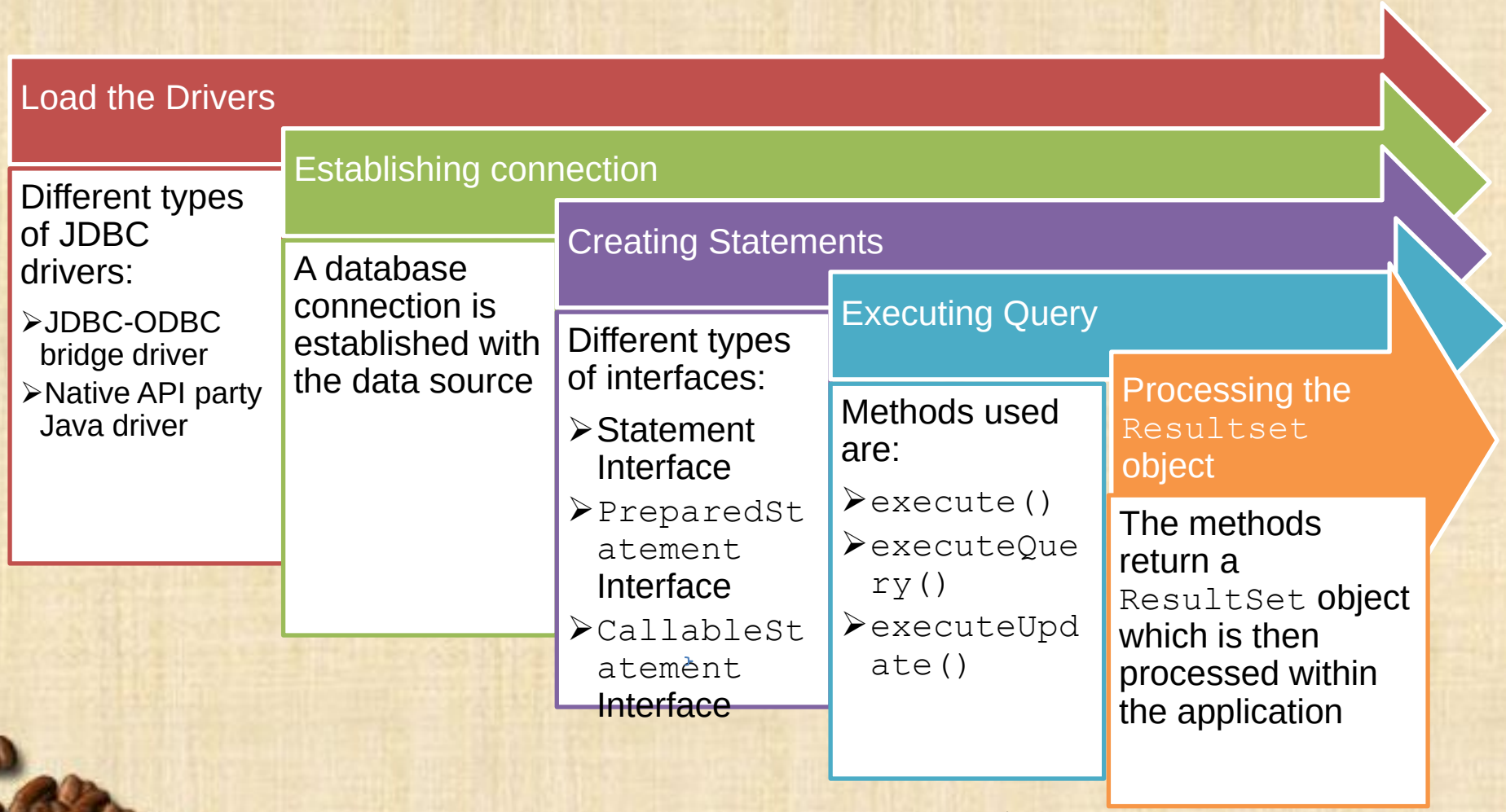


Types of Intercommunication Provided by Resource Adapter

- Outbound Communication
- Inbound Communication
- Bidirectional Communication

JDBC in Action

Steps to process SQL statements using JDBC API:

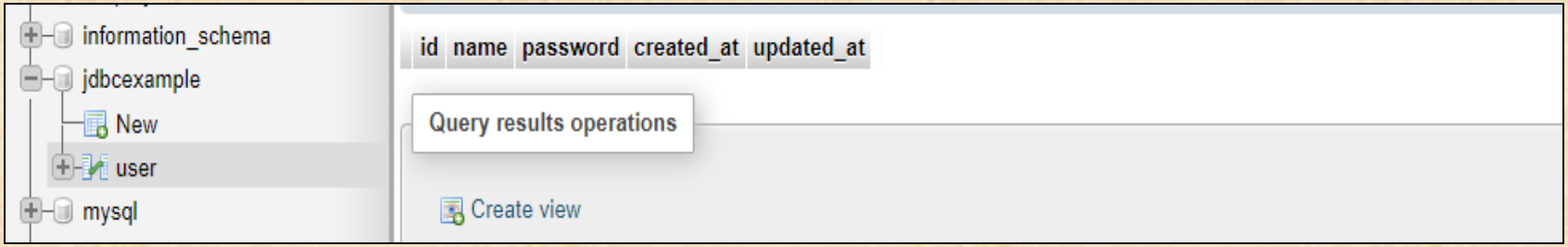


JDBC Example

Code Snippet 1:

```
CREATE TABLE `user` (
 `id` int(10) NOT NULL AUTOINCREMENT,
 `name` varchar(64) NOT NULL,
 `password` varchar(64) NOT NULL,
 `created_at` timestamp NOT NULL DEFAULT current_timestamp() ON UPDATE
current_timestamp(),
 `updated_at` timestamp NOT NULL DEFAULT current_timestamp() ON UPDATE
current_timestamp()
)
```

Successful creation of user table in the MySQL database



JDBC Example for Table User

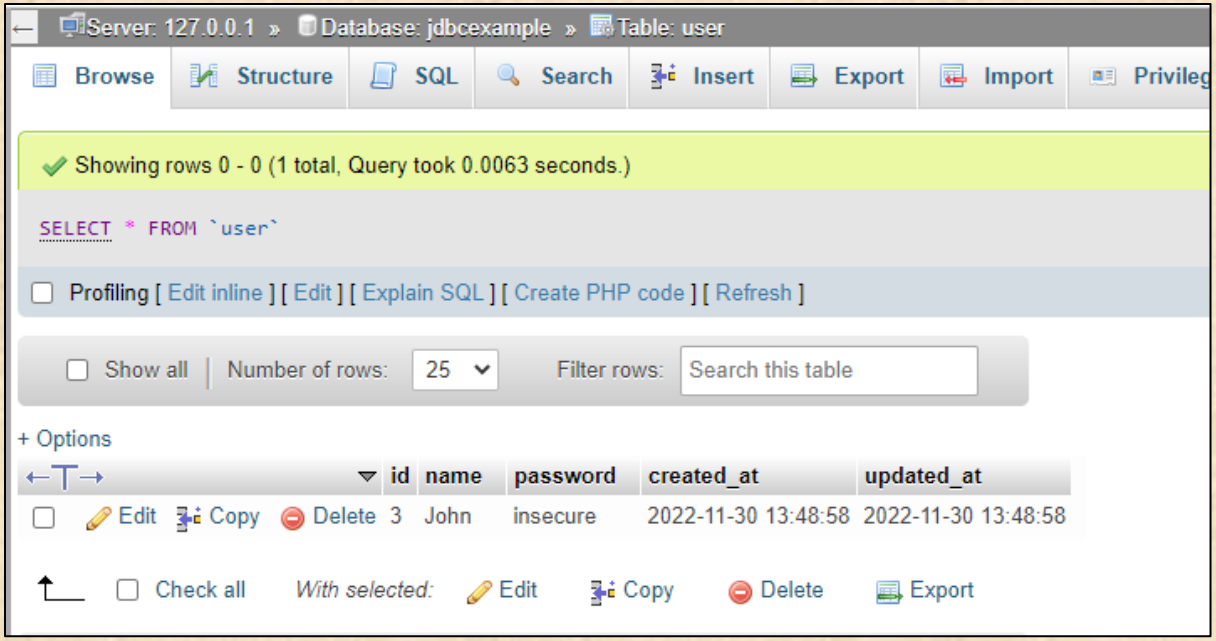
Code Snippet 2:

```
package org.java.mysql.example;
import java.sql.*;

public class JDBC_Example {
 public static void main(String[] args) {
 JDBC_Example main = new JDBC_Example();
 try {
 main.run();
 } catch (Exception e) {
 System.err.println(e.getMessage());
 }
 }

 private void run() throws SQLException,
ClassNotFoundException {
 Connection jdbc_conn = DriverManager.
getConnection("jdbc:mysql://127.0.0.1:3306/jdbc
example", "root","");
Statement jdbc_state =
jdbc_conn.createStatement();
 ResultSet OutputResult =
jdbc_state.executeQuery("SELECT * FROM
jdbcexample.user");
 System.out.println("Updating Table : " +
OutputResult.getMetaData().getTableName(1));
 int columnCount =
OutputResult.getMetaData().getColumnCount();
 for (int i = 1; i <= columnCount; i++) {
 System.out.println(i + " " +
OutputResult.getMetaData().getColumnName(i));
 }
 System.out.println("Searching for table user.");
 boolean exampleUserFound = false;
 while (OutputResult.next()) {
 String username = OutputResult.getString("name");
 if (username.equals("John")) {
 Timestamp createdAt =
OutputResult.getTimestamp("created_at");
 Timestamp updatedAt =
OutputResult.getTimestamp("updated_at");
 System.out.println("Example user found.");
 System.out.println("Name: " + username);
 System.out.println("Created at: " +
createdAt);
 System.out.println("Updated at: " +
updatedAt);
 exampleUserFound = true;
 break;
 }
 }
 if (!exampleUserFound) {
 System.out.println("Inserting a new user.");
 PreparedStatement preparedStatement =
jdbc_conn.prepareStatement("INSERT INTO
jdbcexample.user (name, password,
created_at) VALUES (?, ?, ?)");
 preparedStatement.setString(1, "John");
 preparedStatement.setString(2, "insecure");
 preparedStatement.setTimestamp(3, new
Timestamp(new java.util.Date().getTime()));
 preparedStatement.executeUpdate();
 }
 }
}
```

Output for the JDBC Application

```
Updating Table : user
1 id
2 name
3 password
4 created_at
5 updated_at
Searching for table user.
Inserting a new user.
```

MySQL Table Result After Compilation



Resource Managers and Connections 1-2

Resource Manager

A resource manager is in charge of a group of shared EIS resources.

A transaction processing resource manager can take part in transactions that are controlled and coordinated externally by a transaction manager.

A client of a resource manager within the context of the connector architecture can either be a middle-tier application server or a client-tier application.

A resource manager is generally situated in a distinct address space or on a different device than the client.

Resource Managers and Connections 2-2

Connections

- A connection connects developers to a resource manager.
- It allows an application client to connect to a resource manager, execute transactions, and access the services of resource manager.
- A connection can be transactional or non-transactional in nature.
- Based on the capabilities of the resource manager, a client could establish a connection to a resource manager for bi-directional communication.

Application Components

An application component connects to the enterprise information system using a connection instance it accesses through a connection factory (EIS).

Examples: Database connections, connections to the Java Message Service.

An application component is a server-side component that is deployed, managed, and executed on an application server.

Summary

- The Jakarta Connector architecture specifies a common architecture for coupling heterogeneous EISs with the Jakarta platform.
- The Java SE platform introduced JDBC API specification that allows the developer to persist the data of the object into the relational databases.
- The connector architecture defines a standard SPI to integrate an application server's transaction, security, as well as connection management facilities with those of a transactional resource manager.
- Various resource adapters, one for each type of EIS, can be plugged into an application server.
- An application component connects to the Enterprise Information System using a connection instance it accesses through a connection factory (EIS).

Session: 9

Bean Validation

Jakarta Enterprise Beans in Action

Objectives

- Identify Bean Validation Constraints
- Describe Repeating Annotations
- Explain Validating Constructors and Methods
- Explain Validating Null and Empty Strings
- Elaborate how to create Custom Constraints
- Describe Temporal Constraints Using Clock Provider
- Explain Built-In Value Extractors
- Identify the process of customizing Validator Messages
- Explain Validation of Beans

Bean Validation Constraints 1-2

- A Java specification called Bean Validation facilitates data validation and error detection.
- Validation could be done manually or through integration with other specific requirements and frameworks such as Contexts and Dependency Injection (CDI), JPA, or JSF.
- The `jakarta.validation.constraints` package includes many built-in annotations.
- A custom constraint in Jakarta Bean Validation is created when the user first creates a constraint annotation and then, enforces a constraint validator.

Bean Validation Constraints 2-2

The `@Size` annotations specify that the string value should not be longer than 20 characters.

The `@NotNull` annotation indicates that the address cannot be null.

Code Snippet 1:

```
public class Rental {
@Size(Max=20)
String Home_Builder;
@NotNull @Size(Max=20)
String Home_address;
public String getAddress() {
return Home_address;
}
public String getBuilder() {
return Home_address;
}
public String setAddress(String Home_newAddress)
{
return Home_address = Home_newAddress;
}
public String setBuilder(String Home_Builder) {
return Home_Builder = newHome_Builder;
}
}
```

Repeating Annotation 1-3

Users may implement the same annotation to a declaration or type use. Users can do this with repeating annotations starting with Jakarta.

The `@Repeatable` meta-annotation must be applied to the annotation type.

The value of the `@Repeatable` meta-annotation is the type of container annotation generated by the Java compiler to store repeating annotations.

Repeating `@Schedule` annotations are stored in a `@Schedules` annotation.

Repeating Annotation 2-3

Using `@Repeatable` to print the schedule.

Code Snippet 2:

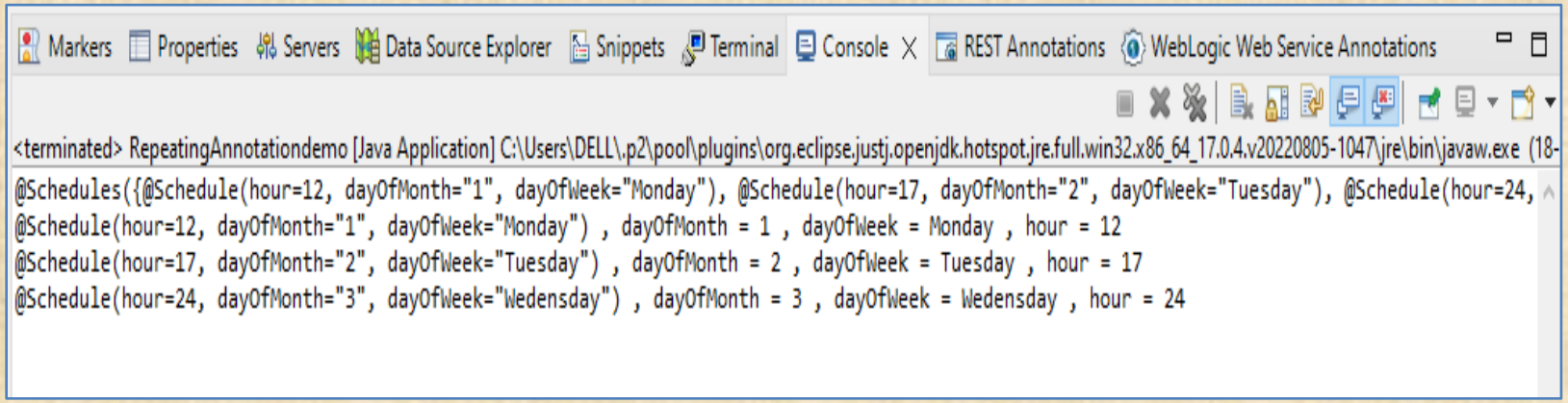
```
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import
java.lang.annotation.RetentionPolicy;
@Repeatable(Schedules.class)
@interface Schedule {
String dayOfMonth() default "1";
String dayOfWeek() default "Monday";
int hour() default 12;
}
@Retention(RetentionPolicy.RUNTIME)
@interface Schedules {
Schedule[] value();
}
@Schedule
@Schedule(dayOfMonth = "2",dayOfWeek =
"Tuesday", hour = 17)
@Schedule(dayOfMonth = "3", dayOfWeek =
"Wedensday", hour = 24)
public class ServerRestartSchedule
{
}
```

Code Snippet 3:

```
public class
RepeatingAnnotationdemo {
public static void main(String[]
args)
{
Schedules schedules =
ServerRestartSchedule.class.getAnno
tation(Schedules.class);
System.out.println(schedules);
Schedule[] scheduleArray =
schedules.value();
for (Schedule schedule :
scheduleArray) {
System.out.println(schedule + " ,
dayOfMonth = "+
schedule.dayOfMonth() + " ,
dayOfWeek = "
+ schedule.dayOfWeek() + " , hour =
" + schedule.hour());
}
}
}
```

Repeating Annotation 3-3

Output of Code Snippet 3



Validating Constructors and Methods

Bean Validation constraints can be applied to nonstatic method and function constructor parameters, as well as nonstatic method return values.

Code Snippet 4:

```
public class Comp_Employee {
public Employee (@NotNull String name) { ... }
public void setSalary(
@NotNull
@Digits(integer=6, fraction=2) BigDecimal
salary,
@NotNull
@ValidCurrency
String currencyType) {
...
}
...
}
```

Code Snippet 5:

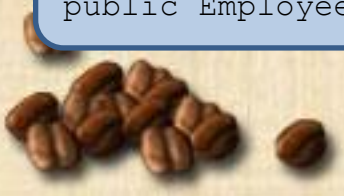
```
@ConsistentPhoneParameters
@NotNull
public Employee (String name,
String officePhone, String
mobilePhone) {
...
}
```

Code Snippet 6:

```
@NotNull
public Employee getEmployee() { ... }
```

Code Snippet 7:

```
@Manager(validationAppliesTo=ConstraintTarget.RETURN_VALUE)
public Employee getManager(Employee employee) { ... }
```



Validating Null and Empty Strings

An empty string is a zero-length string example, while a null string has no value at all.

An EJB cannot perform any function unless it is invoked by a client.

Code Snippet 8:

```
if (testString==null) {
doSomething();
} else {
doAnotherThing();
}
```

Code Snippet 9:

```
<context-param>
  <param-name>
javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
  <param-name>
  <param-value>true/<param-value>
</context-param>
```

Creating Custom Constraints

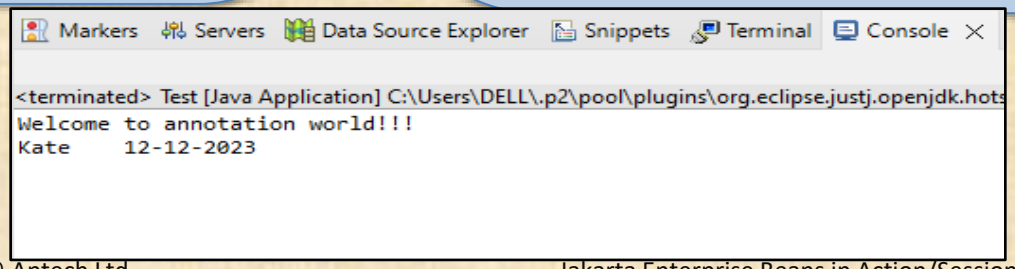
Code Snippet 10:

```
package org.com.client;
import
java.lang.annotation.Documented;im
port
java.lang.annotation.Retention;
import
java.lang.annotation.RetentionPoli
cy;
//user-defined or custom
annotation
@Documented
@Retention(RetentionPolicy.RUNTIME
)
public @interface MyAnnotation
{
public abstract String Name()
default "John Kate";
public abstract String Date();
}
```

Code Snippet 11:

```
package org.com.client;
import java.lang.reflect.Method;
import org.com.client.MyAnnotation;
public class Test {
@MyAnnotation(Name = "Kate", Date = "12-12-
2023"
public void myMethod1() {
System.out.println("Method1");
}
public static void main(String args[])
throws NoSuchMethodException,
SecurityException {
System.out.println("Welcome to annotation
world!!!");
Method method = new
Test().getClass().getMethod("myMethod1");
MyAnnotation annotation =
method.getAnnotation(MyAnnotation.class);
System.out.println(annotation.Name()+"\t"+an
notation.Date());
}
}
```

Output of Code Snippet 11



Temporal Constraints Using Clock Provider

- Constraint validators for temporal constraints can retrieve the current instant from the `ClockProvider` object returned by `ConstraintValidatorContext#getClockProvider()`.

Code Snippet 12:

```
public interface ClockProvider {
Clock getClock();
}
```

- The method `getClock()` returns `java.time`.

Built-in Value Extractors

Suitable deployments provide value extractors for following types:

- `iterable; java.util.Iterable Value()` must be called for each contained element, with the string literal serving as the node name.
- `java.util.Map`; support for both map keys and map values.
- `java.util.List value()` must be called for each contained element, with the string literal serving as the node name.
- `java.util.Optional; value()` must be called, with null as the node name and the contained object as the value, or null if none is present.

Customizing Validator Messages

- Bean Validation comes with a set of default messages for the built-in constraints.
- `ValidationMessages.properties` locale variations are appended by appending an underscore and the locale prefix to the base name of the file.
- The `ValidationMessages` resource package, as well as its locale variants, encompass strings that override the default validation messages.
- The `ValidationMessages` resource package is generally a properties file.

Validation of Beans

Bean validation is one of the most common methods for validating input in Java.

One advantage of this approach is that the validation constraints and validators are only written once, which reduces duplication of effort and ensures consistency.

Summary

- Jakarta Bean Validation, formerly known as Java Bean Validation, by using annotations enables user to specify data validations in Java Beans.
- Validating user input to verify the integrity is an essential part of application logic. Data validation can occur at various layers in even the most basic applications.
- The Jakarta Bean Validation API describes a collection of common constraint annotations, such as `@NotNull` and `@Size`.
- In order to create a custom constraint in Jakarta Bean Validation, the user must first create a constraint annotation and then, enforce a constraint validator.
- Bean Validation constraints can be applied to nonstatic method and function constructor parameters, as well as nonstatic method return values.
- Constraint validators for temporal constraints (either built-in constraints `@Past`, `@PastOrPresent`, `@Future`, and `@FutureOrPresent`, or custom temporal constraints) can retrieve the current instant from the ClockProvider object returned by `ConstraintValidatorContext#getClockProvider()`.
- Bean Validation comes with a set of default messages for the built-in constraints. Such messages can be personalized and localized.

Session: 10

Execution of Enterprise Beans Using Transactions

Jakarta Enterprise Beans in Action

Objectives

- Explain how to use of transactions in Enterprise Beans
- Describe different events taking place
- Explain Java Naming and Directory Interface (JNDI)
- Elaborate on Remote/Distributed execution

Introduction

- A group of operations that are intended to be carried out jointly constitute a transaction.
- For example, a purchase through E-commerce Website requires a number of steps.
- Transactions guarantee the accuracy of the data kept in the database.

Transactional Operations

- Begin: Start of Transaction
- Commit: End of Transaction
- Rollback: Return to pervious stable state

Difference between Programmatic and Declarative transaction demarcation methods:

Programmatic Transaction Demarcation	Declarative Transaction Demarcation
1. Used when there are few transactional operations.	1. Beneficial when there are lot of transactions.
2. Programmer decides start and end of transaction.	2. Boundaries are unidentified within the code.

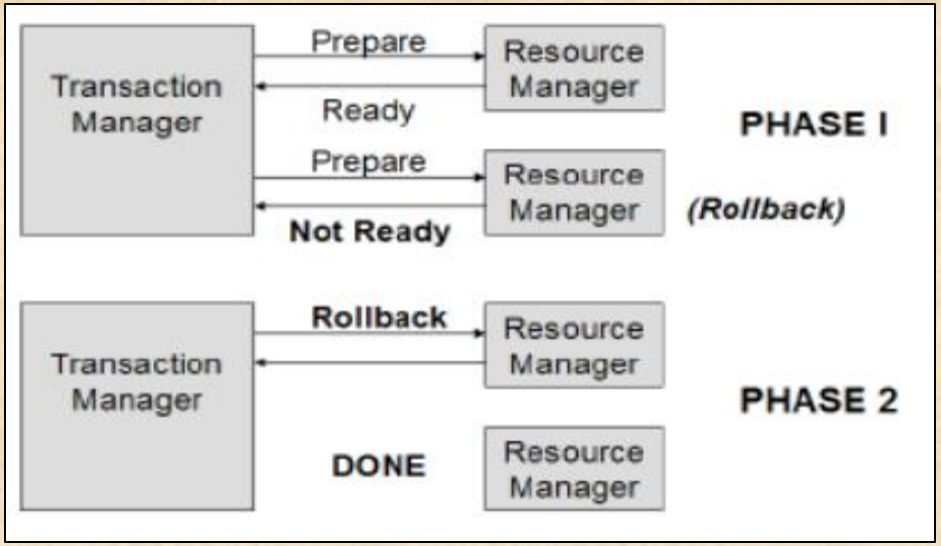
Transaction Processing Components

Application Components: These are the EJB business methods that invoke transactions.

Resource Managers: These components manage the persistent data storage. They are usually drivers with interfaces for communicating with databases.

Transaction Manager: This is the core component that creates and maintains transactions.

Transaction Manager commits a transaction in two phases:



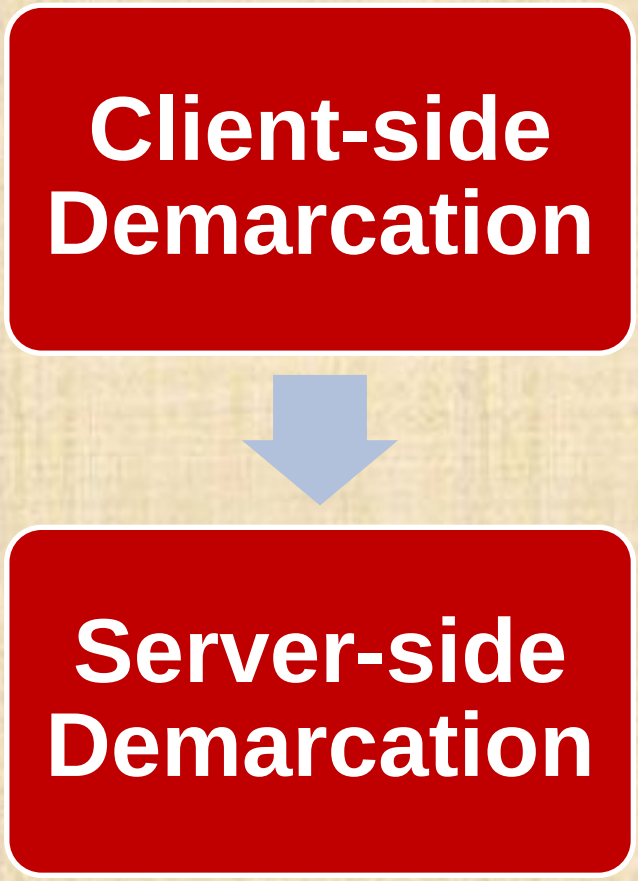
Transaction Outcomes

- A transaction is considered successful if it is neither cancelled nor terminated due to a hazard.
- If an execution reaches the cancel end event, the transaction is cancelled.
- A hazard terminates a transaction if an erroneous event is thrown that is not captured inside the scope of the transaction subprocess.

For Example: ATM Transaction

Login using valid credentials. → Enter the amount to be withdrawn. → Check for sufficient balance in the account. → The ATM dispenses cash.

Java Naming and Directory Interface (JNDI)



Retrieving a `UserTransaction` object from the Namespace

Code Snippet 1:

```
ic = new InitialContext ( );

// lookup the user transaction object
UserTransaction ut = (UserTransaction)ic.lookup
("java:comp/UserTransaction");
ut.begin ();
...
ut.commit();
```

Steps to Demarcate the Transaction

1. Create a `Hashtable` environment, fill it with the namespace address and authentication information.
2. Within the client logic, retrieve the `UserTransaction` object from the namespace.
3. Begin the global transaction in the client by calling `UserTransaction.begin()`.
4. Get the server Bean.
5. Call any object methods that will be included in the transaction.
6. Complete the transaction by calling `UserTransaction.commit()` or `UserTransaction.rollback()`.

Remote/Distributed Execution

Within the EJB container, distributed transactions can be of two types:

Container-managed transactions

Bean-managed transactions

JDBC Local Transactions

A JDBC application uses local transactions to make changes to a database permanent and to indicate the end of a unit of work in one of the ways described as follows:

After executing one or more SQL statements, call the `Connection.commit` or `Connection.rollback` methods.

At the start of the application, call the `Connection.setAutoCommit(true)` method to commit changes after each SQL statement.

Applications participating in distributed transactions, cannot use `the Connection.commit, Connection.rollback, or Connection.setAutoCommit(true)` methods within the distributed transaction.

Transaction in a Cart

Code Snippet 2: CartBean

```
package com.session10.beans;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
import jakarta.util.ArrayList;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Remove;
import javax.ejb.Stateless;
import jakarta.ejb.TransactionManagement;
import jakarta.ejb.TransactionManagementType;
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import jakarta.ejb.AfterBegin;
import jakarta.ejb.AfterCompletion;
import jakarta.ejb.BeforeCompletion;
@Stateless
@TransactionManagement(value=TransactionManagementType.CONT
AINER)
@LocalBean
public class CartBean  {
    private ArrayList Cart_items;

    @PostConstruct
    public void init() {
        Cart_items = new ArrayList();
        System.out.println("Cart functionality start:
init");
    }
    @PreDestroy
    public void destroy() {
        System.out.println("Cart functionality: destroy");
    }

    @Remove
    public void checkOut() {
        // Release any resources.
        System.out.println("Cart checkout...");
    }
```

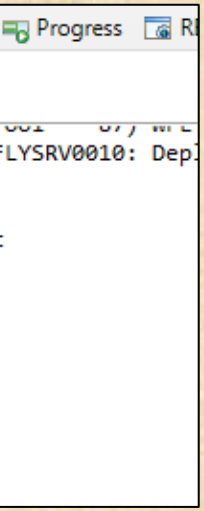
```
public void addItem(String item) {
        getItems().add(item);
        System.out.println(item + " inserted into
cart");
    }

public void removeItem(String item) {
        getItems().remove(item);
        System.out.println(item + " item removed from
cart");
    }

    public ArrayList getItems() {
        return Cart_items;
    }

    @AfterBegin
    private void afterBegin(){
        System.out.println("A new transaction has
started.");
    }

    @BeforeCompletion
    private void beforeCompletion(){
        System.out.println("You're about to commit to a
transaction.");
    }

    @AfterCompletion
    private void afterCompletion(boolean committed) {
        System.out.println("Once a transaction commit
protocol has finished, it notifies the instance based on
the committed value whether the transaction has been
committed or rolled back: " + committed);
    }
     }
```

Code Snippet 3: No_TX_Client_Tester Class

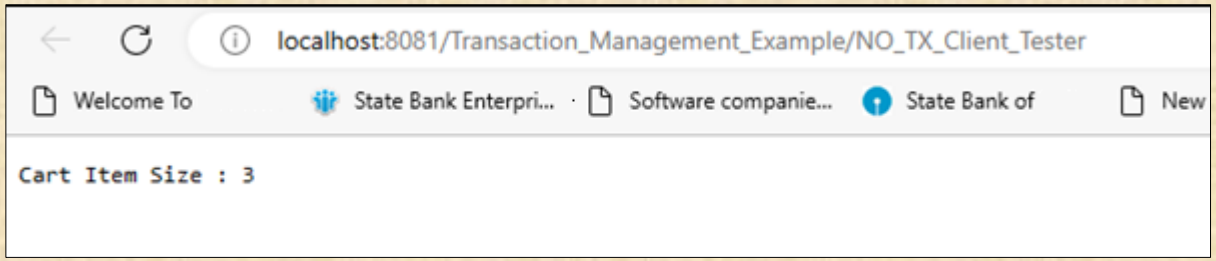
```
package com.session10.non_tx;
import jakarta.io.IOException;
import jakarta.io.PrintWriter;
import jakarta.util.logging.Level;
import jakarta.util.logging.Logger;
import jakarta.naming.Context;
import jakarta.naming.InitialContext;
import jakarta.naming.NamingException;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import com.session10.beans.CartBean;

@WebServlet(name = "NO_TX_Client_Tester", urlPatterns =
{"/NO_TX_Client_Tester"})
public class NO_TX_Client_Tester extends HttpServlet {


    protected void processRequest(HttpServletRequest
request,
HttpServletResponse response)
            throws ServletException, IOException {
        try (PrintWriter out = response.getWriter()) {
            CartBean cartBean = lookupCartBeanBean();
            cartBean.addItem("LCD TV");
            cartBean.addItem("Smart Watch");
            cartBean.addItem("Iphone 15 pro");
            out.println("Cart Item Size : " +
cartBean.getItems().size());
            cartBean.checkOut();
        }
    }
```

```
    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest
request,
HttpServletResponse response)
  throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    public String getServletInfo() {
        return "Short description";
    }
    private CartBean lookupCartBeanBean() {
        try {
            Context c = new InitialContext();
            return (CartBean)
c.lookup("java:global/TestTransaction1/CartBean
!com.session10.beans.CartBean");
        } catch (NamingException ne) {
Logger.getLogger(getClass().getName()).log(
Level.SEVERE, "exception caught", ne);
 throw new RuntimeException(ne);
        }
    }
}
```

Output of the Cart Application on the Console



Output of the Cart Application at Browser Side



Summary

- A transaction can be defined as a group of operations that are supposed to be executed as a single unit.
- EJB supports Programmatic and declarative transaction demarcation methods.
- The transaction management mechanism in the middle layer is provided by the Java Transaction API (JTA).
- The EJB specifications refer to the declarative transactions as Container-Managed Transaction (CMT) and programmatic transactions as Bean-Managed Transaction (BMT).
- The transactions in Java applications are managed through Java Transaction Service (JTS) and JTA.
- JTA is an interface between a transaction manager and the components involved in a distributed transaction system. The transactions of an application are managed by the JTA API in the container.

Session:11

Jakarta Context and Dependency Injection (CDI)-I

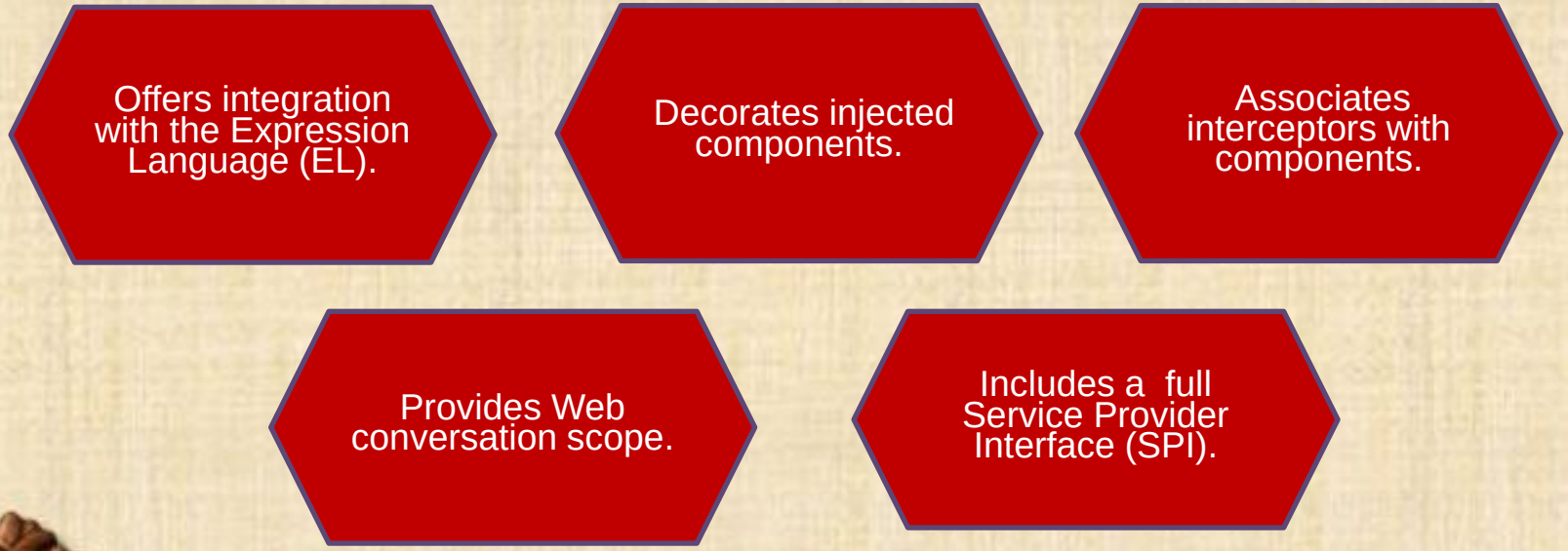
Jakarta Enterprise Beans in Action

Objectives

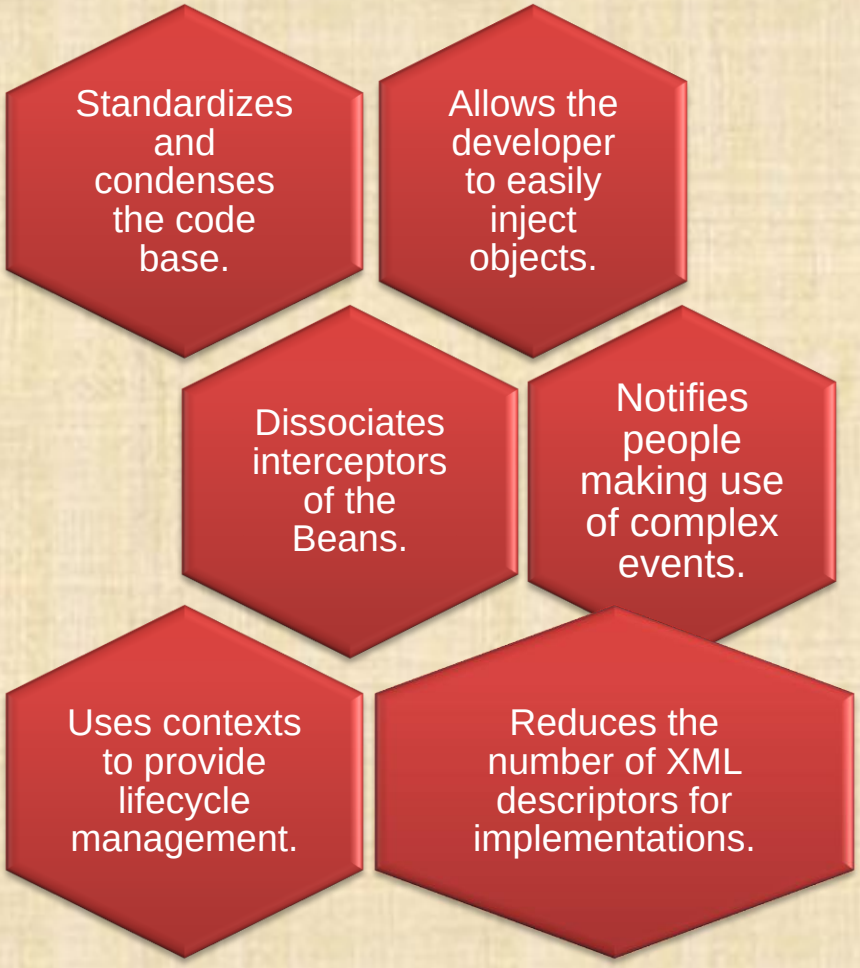
- Describe an overview of Jakarta Context and Dependency Injection
- List and describe the benefits of Jakarta Context
- Explain how to enable Development Mode
- Identify the purpose of the default Bean Discovery mode
- Outline annotations used in Bean Context
- Explain how to exclude Beans from the scanning process

Jakarta Context

- Using Context, developers associate stateful component interactions and lifecycles with predefined lifecycle contexts.
- Using Dependency injection service, at deployment time implementation of an interface to be injected can be chosen.
- Features of CDI:



Advantages of Dependency Injection and Jakarta Context



Enable Development Mode

- Following techniques can be used to enable CDI Development Mode:
  1. In `web.xml`, the application specifies a context parameter
  2. At deployment time, enabled mode (only for individual applications) is made possible by a system property

- Following context parameter can be added to the `web.xml` descriptor:

```
<web-app>
...
    <context-param>
        <param-name>org.jboss.weld.development</param-name>
        <param-value>true</param-value>
    </context-param>
...
</web-app>
```

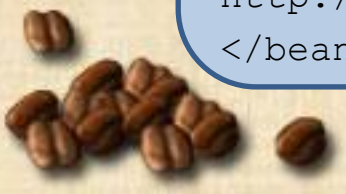
Default Bean Discovery

- Two variants for bean archive files are:
  1. Implicit bean archive files
  2. Explicit bean archive files

- Implicit bean archive contains some beans annotated with a scope type. It does not contain `beans.xml` deployment descriptor.
- An explicit bean archive comprises a `beans.xml` deployment descriptor.

Code Snippet 1:

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



Annotations Used in Bean Context

- `@Resource`, `@PostConstruct`, and `@PreDestroy` are examples of annotations.
- The `@Bean` annotation allows developers to specify any initialization and destruction callback methods.
- The `@Scope` directive allows the developer to override the singleton scope that is used by default.
- A class that is `@Configuration`- or `@Component`-annotated can use the `@Bean` annotation.

Example Showing Use of Annotations 1-5

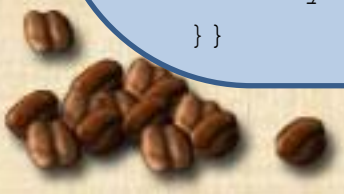
Code Snippet 2:

```
package com.session11.co.ExampleCalculator;
public class Calculator {
    public int Sum(int x, int y) {
        return x + y;
    }
    public int Sub(int x, int y) {
        return x - y;
    }
    public int Mul(int x, int y) {
        return x * y;
    }
}
```

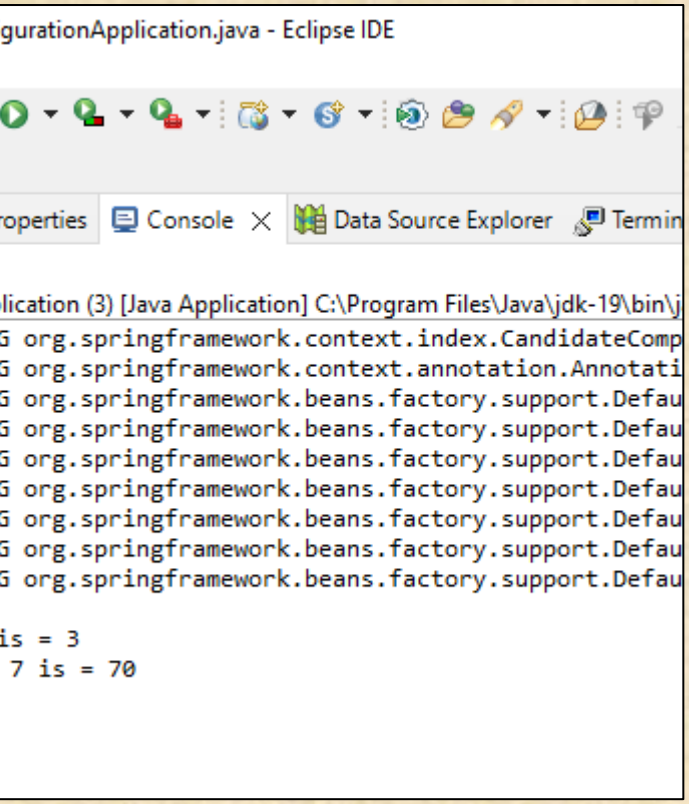
Example Showing Use of Annotations 2-5

Code Snippet 3:

```
package com.session11.co.ExampleCalculator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
@Configuration
public class ConfigurationApplication {
    @Bean
    Calculator calculator() {
        return new Calculator();
    }
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =    new
AnnotationConfigApplicationContext(ConfigurationApplication.class);
        Calculator ConfigurationApplication. Class =
context.getBean(Calculator.class);
        int SumVal = calculator.Sum(10, 7);
        int SubVal = calculator.Sub(10, 7);
        int MulVal = calculator.Mul(10, 7);
        System.out.println("Sum of 10 and 7 is = " +SumVal);
        System.out.println("Subtraction of 10 and 7 is = " +SubVal);
        System.out.println("Multiplication of 10 and 7 is = "+MulVal);
   }}
```



Example Showing Use of Annotations 3-5



Example Showing Use of Annotations 4-5

Code Snippet 4:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.session11.co</groupId>
  <artifactId>ExampleCalculator;
</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.3.23</version>
    </dependency>
  </dependencies>
```

Example Showing Use of Annotations 5-5

```
<build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-
plugin</artifactId>
        <version>3.10.1</version>
        <configuration>
          <source>8</source>
          <target>8</target>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Excluding Beans From Scanning Process

Code Snippet 5:

```
<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/jakarta">
   <scan>
      <exclude name="com.acme.swing.**" />
<!—Don't include GWT support if GWT is not installed -->
      <exclude name="com.acme.gwt.**">
         <if-class-not-available name="com.google.GWT"/>
      </exclude>
<!--
Exclude types from com.acme.verbose package if the
system property verbosity is set to low
That is, jakarta ... -D verbosity=low
-->
      <exclude name="com.acme.verbose.*">
<if-system-property name="verbosity" value="low"/>
      </exclude>
```

```
 <exclude name="com.acme.jsf.**">
        <if-class-
available name="org.apache.wicket.Wicket"/>
        <if-system-property name="viewlayer"/>
      </exclude>
   </scan>
</beans>
```

Summary

- Using Jakarta Context service, developers can build stateful component interactions and lifecycles with predefined, but extensible lifecycle contexts.
- Using the Dependency injection service, users can choose which implementation of a specific interface is to be injected into an application.
- The Jakarta EE environment or even other environments that support CDI could be used to implement an application that makes use of services.
- A bean archive that is annotated with its default bean discovery mode is referred to as an implicit bean archive.
- When developing and inspecting applications that contain CDI Beans, CDI Development Mode makes some CDI-related tools and features available.
- Developers can exclude classes from the archive from being scanned, having container lifecycle events fired, and being deployed as Beans CDI Beans.

Session:12

Jakarta Context and Dependency Injection (CDI)-II

Jakarta Enterprise Beans in Action

Objectives

- Explain features of Qualifiers and Injecting Beans
- Describe how to use Scopes
- Explain how to apply Setter and Getter Methods
- Explain constructing a Managed Bean in a Jakarta Facelet Page
- Describe how to configure a CDI Application

Importance of Dependencies

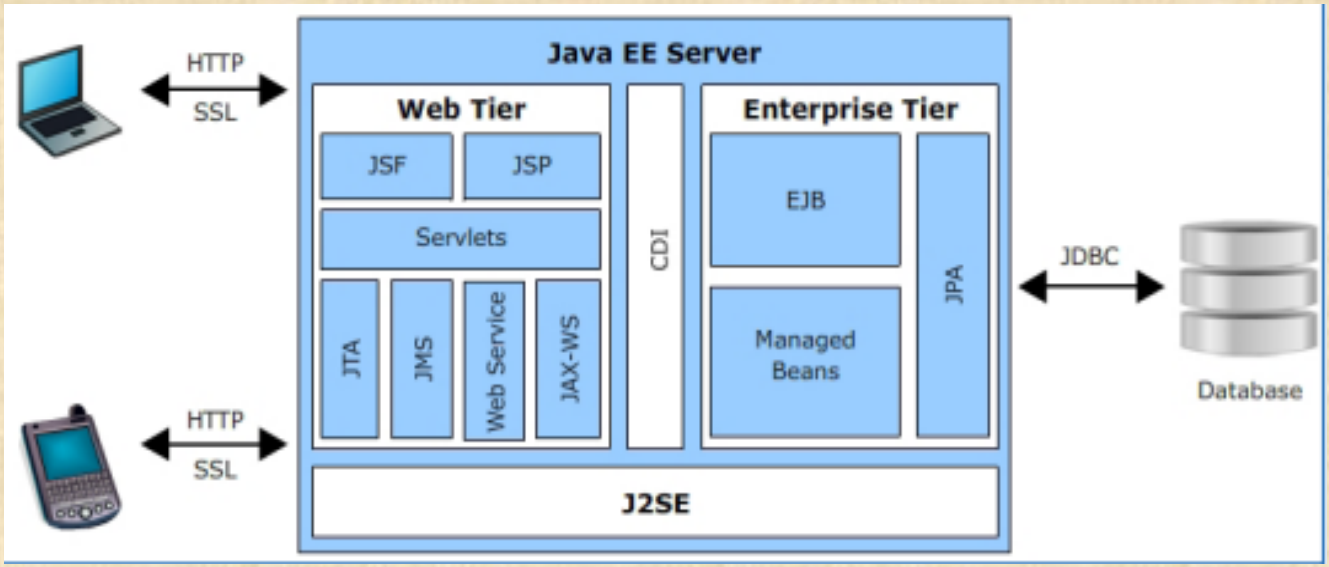
DI is a method of developing software in which different components are loosely coupled.

Users can use DI to transform regular Java classes into managed objects and inject them into virtually any other managed object.

Annotations in DI allow application components to declare dependencies on external resources and configuration parameters.

Jakarta Context Dependency Injection

- CDI is a set of services that make it easy for developers to use Enterprise Beans along with Jakarta Server Faces technology in Web applications.
- CDI allows the EJBs to be used as managed beans for JSF applications. This helps the Web tier to interact directly with the business and persistence tiers.
- **Interrelationship among Jakarta EE Specifications**



Features of Qualifiers and Injecting Beans

Objects which CDI enables to be injected into applications:

- CDI allows any Jakarta class to be injected as a resource into the application.
- CDI enables injection of Session beans as resources.
- CDI also enables injection of Jakarta EE resources such as data sources, Jakarta Message Service topics, queues, and so on.
- CDI allows injection of persistence contexts and also producer fields.

Usage of @Inject Annotation 1-2

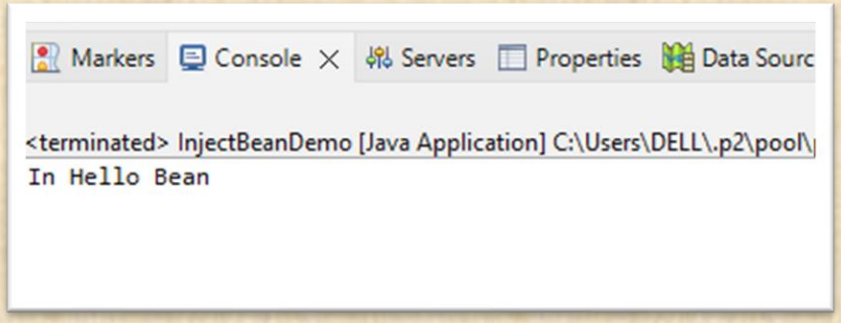
Code Snippet 1: InjectBeanDemo.java

```
package demo.inject;

import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
import jakarta.inject.Inject;

/**
 * Session Bean implementation class InjectBeanDemo
 */
@Stateless
@LocalBean
public class InjectBeanDemo {
        @Inject
        private HelloBean H;
        public InjectBeanDemo(){
        H= new HelloBean();
        }
        public static void main(String args[]){
        InjectBeanDemo I = new InjectBeanDemo();
        }
        }
```

Usage of @Inject Annotation 2-2

Code Snippet 2: HelloBean.java

```
package demo.inject;
import jakarta.ejb.LocalBean;
import jakarta.ejb.Stateless;
/**
 * Session Bean implementation class HelloBean
 */
@Stateless
@LocalBean
public class HelloBean {
    public HelloBean() {
        System.out.println("In Hello Bean");
    }
}
```



Using Scope

The scope of a session bean can have any value from adapted ones:

- Request
- Session
- Application
- Dependent
- Conversation

Adding Setter and Getter Methods with Beans

- To start making the managed state of the bean approachable, developers can add Setter and Getter methods for that state.
- In Java, Getters and Setters are two methods for retrieving and upgrading the values of a variable.
- Getter methods are responsible for retrieving the most recent value of a variable.
- Setter methods are responsible for setting or updating the value of an existing variable.
- Getter and Setter methodologies are also known as accessors and mutators respectively.

Using Managed Bean in a Jakarta Facelet Page

A Managed Bean is a specialized Jakarta class that synchronizes value systems with elements, processes business logic, and manages page navigation.

Managed Beans are used by Jakarta Server Faces to differentiate presentation from business logic.

The bean execution code contains the program logic and Jakarta Server Faces.

In Jakarta Server Faces, each element has a collection of attributes for referencing the managed bean technique.

Configuring a CDI Application

- The server recognizes the application as a bean archive when beans are annotated with a scope type.
- `Beans.xml` is an optional deployment descriptor used by CDI.
- Various `Beans.xml` configuration settings are used over and above annotation settings in CDI classes.
- In the event of a conflict, the configurations in `beans.xml` take precedence over the annotation settings.
- Only within a few cases must an archive include the `beans.xml` deployment descriptor.

Summary

- Developers can use dependency injection to transform regular Java classes into managed objects and inject them into virtually any other managed object.
- In Jakarta EE, dependency injection characterizes scopes, which govern the lifecycle of the objects that the container instantiates and injects.
- The scope of the bean determines the lifecycle and accessibility of its instances. The CDI context model is expandable and can support arbitrary scopes.
- In Jakarta, Getters and Setters are two methods for retrieving and upgrading the values of a variable.
- Managed Beans are used by JSF to differentiate presentation from business logic.
- The bean execution code contains the program logic, and JSF simply refers to the bean characteristics or actions that use the Expression Language (EL).
- The server recognizes the application as a bean archive once the beans are annotated with a scope type. No extra configuration is required.

Session: 13

CDI Beans

Jakarta Enterprise Beans in Action

Objectives

- Explain how to create CDI Bean Qualifiers
- Explain how to create Producers, Disposers, Interceptors, Events, and Stereotypes
- Describe Jakarta EE Declarative and Programmatic Approach
- Explain Security and Configure Authentication
- Describe basics of using Application roles, Security Constraints, and Login Modules

Creating CDI Bean Qualifiers

- CDI is a set of services that makes it easy for developers to work with Enterprise Beans.
- CDI gives a great deal of flexibility to integrate different types of components in a typesafe manner.
- The container controls lifecycles of the Beans and can inject beans into other beans.
- Two main approaches for injecting beans into other beans are:
  - Employing annotations
  - Making use of the `beans.xml` file

Using Qualifiers

- Qualifiers are user-defined annotations.
- Developers can specify the bean implementation version they want to use at runtime by using a Qualifier.
- The Qualifier must be defined and annotated on both bean and the injection point.

Implementation of Qualifier by Autowiring 1-4

- An example for Qualifier implementation for printing Employee address using autowiring and injecting method.

Code Snippet 1 : `Address` **class**

```
package com.session13.co.DemoQualifierTest;

public class Address {
private String street;
private String city;
public String getStreet() {
            return street;
}
public void setStreet(String street) {
            this.street = street;
}
public String getCity() {
            return city;
}
public void setCity(String city) {
            this.city = city;
}
@Override
public String toString() {
            return " [street=" + street + ", city=" + city + "]";
}
}
```

Implementation of Qualifier by Autowiring 2-4

- Any class annotated with `@Component` or methods annotated with `@Bean` can use `@Qualifier` annotation.

Code Snippet 2 : `Emp` **class**

```
package com.session13.co.DemoQualifierTest;

import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.beans.factory.annotation.Qualifier;

public class Emp {

@Autowired
@Qualifier("temp1")
private Address address;

public Address getAddress() {
            return address;
}


public void setAddress(Address address) {
            System.out.println("Setting value");
            this.address = address;
}
```

```
public Emp() {
                super();
                // TODO Auto-generated constructor
stub
}
public Emp(Address address) {
                super();
                this.address = address;
                System.out.println("inside
constructor");
}

@Override
public String toString() {
                return "Employee John [address=" +
address + "]";
}
}
```

Implementation of Qualifier by Autowiring 3-4

Test class acts as Main class which displays the employee address.

Code Snippet 3: `Test` class

```
package com.session13.co.DemoQualifierTest;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

         ApplicationContext context = new
ClassPathXmlApplicationContext("com/session13/co/DemoQualifierTest/autoconfig.xml");
         Emp emp1 = context.getBean("emp1", Emp.class);
         System.out.println(emp1);
}
}
```

Implementation of Qualifier by Autowiring 4-4

- Given Code Snippet enables auto component scanning and is created for configuring Bean.

Code Snippet 4: `autoconfig.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
    <context:annotation-config />
<bean class=" com.session13.co.DemoQualifierTest.Address"  name="temp1">
          <property name="street" value="Park Ave" />
          <property name="city" value="NewYork" />
</bean>
<bean class="com.session13.co.DemoQualifierTest.Emp" name="emp1"   />
</beans>
```

Output:

```
14:47:03.490 [main] DEBUG o.s.c.e.PropertySourcesPropertyResolver - Could not find key 'spring.liveBeansView.mbeanDomain' in any
14:47:03.491 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance of singleton bean 'emp1'
Employee John [address= [street=Park Ave, city=NewYork]]
```

Creating Producers, Disposers, Interceptors, Events, and Stereotypes 1-6

Producers:

- A Producer method in CDI creates an object that can be injected later.
- Producer methods are used in following situations:
  - While injecting a non-bean object.
  - When the concrete type of the object to be injected changes at runtime.
  - When the object requires a custom initialization that the bean constructor does not handle.
- Producer method allows polymorphism.

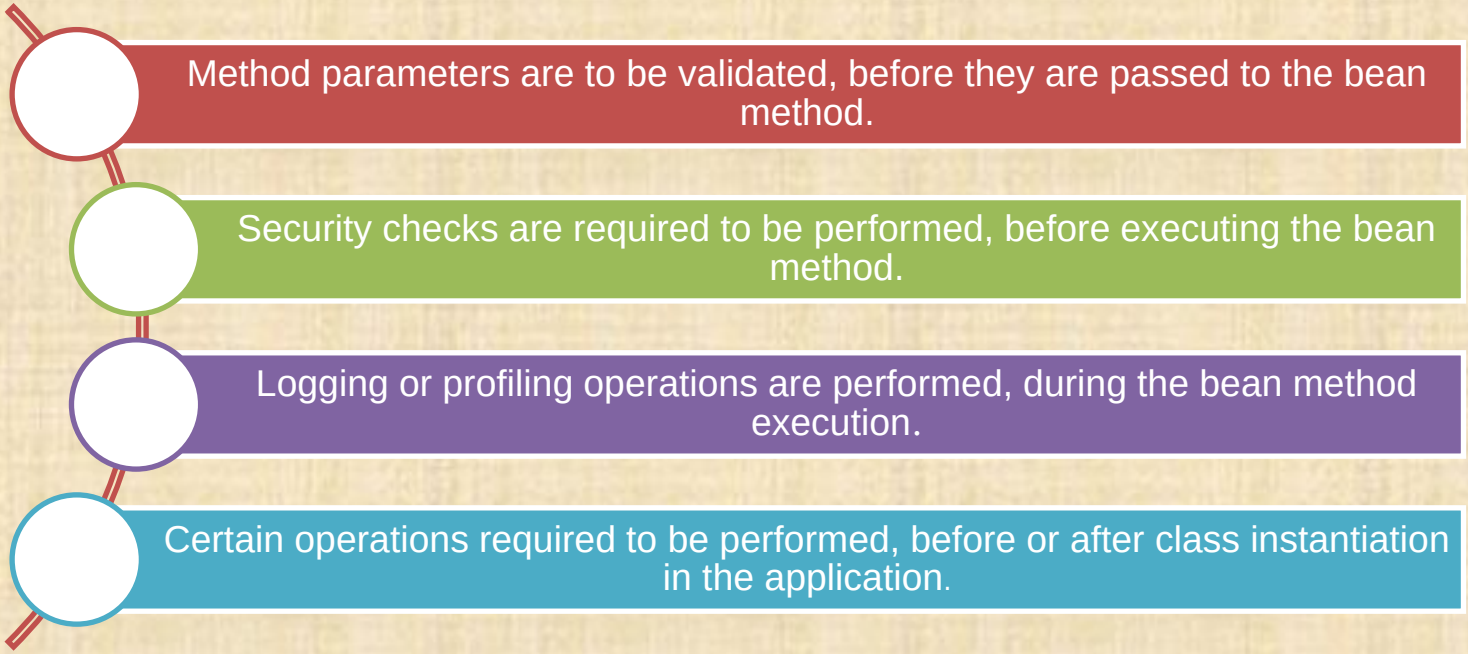
Creating Producers, Disposers, Interceptors, Events, and Stereotypes 2-6

Interceptors:

- Interceptors allow developers to interpose EJB business method, thereby allowing to add wrapper code.
- Wrapper code can be executed before or after the method is called.
- Interceptors provide control on the lifecycle methods of a bean.
- Interceptors are used in association with enterprise bean classes to allow invocation of interceptor methods on the associated target class.

Creating Producers, Disposers, Interceptors, Events, and Stereotypes 3-6

- On Jakarta EE platform, interceptors can be used with Session beans or Message driven beans.
- Scenarios in which interceptors can be executed:



Creating Producers, Disposers, Interceptors, Events, and Stereotypes 4-6

Interceptors can be defined for a target class in two different ways:

Within the target class:
The target class is the Bean class. The methods of the target class that has to be intercepted are annotated with `@jakarta.Interceptor.Interceptors`. The bean can impose the `interceptor` class either on all methods or explicit methods.

Separate Interceptor Class:
The interceptor class contains methods that can be invoked along the lifecycle `callback` methods of the target class or business methods of target class.
Only one `interceptor` class can be defined for a target class.

Creating Producers, Disposers, Interceptors, Events, and Stereotypes 5-6

Events

- Events help in enabling bean interaction completely independent of compile time.
- Producers of events create events, which are delivered in containers to observers of events.
- Observers can choose a variety of selectors to limit the number of event notifications they receive.
- Observers have the option of receiving the event notification immediately or after the current transaction has been completed.

Creating Producers, Disposers, Interceptors, Events, and Stereotypes 6-6

Stereotypes

- A stereotype enables a framework developer to declare some standard metadata for beans in a single location.
- Stereotype is a combination of default scope and a set of interceptor bindings.
- A bean can have zero, one or multiple stereotypes. Annotation with `@Stereotype` bundles several other annotations.
- Some MVC frameworks use stereotype as shown

```
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
…
public @interface Action {}
Developers use the
stereotype by implementing
the annotation to a bean.
@Action
public class LoginAction {
… }
```

Jakarta EE Declarative and Programmatic Approach 1-2

- A multi-tier Enterprise application is constructed using many elements that are in combination.
- Applications of the web and enterprise are composed of parts that are deployed in different containers.
- Containers for the components offer declarative and programmatic security.
- The Jakarta EE security environment allows security constraints to be defined at deployment time.
- An agreement between application component provider and deployer is made using Jakarta annotations and deployment descriptors.

Jakarta EE Declarative and Programmatic Approach 2-2

- Jakarta EE offers two types of approach to ensure security of the application namely:

Declarative Approach

Programmatic Approach

Security and Authentication 1-4

- Enterprise applications contain different components and when they are accessed through internet, have to be protected from unwanted access.
- To ensure security, following steps must be taken care while building the application:

Users must be appropriately authenticated and authorized.

All application components are deployed on the application server and managed logically through the container.

The container provides security services for the components deployed in it.

In Jakarta EE, the security policy is implemented through annotations and deployment descriptors.

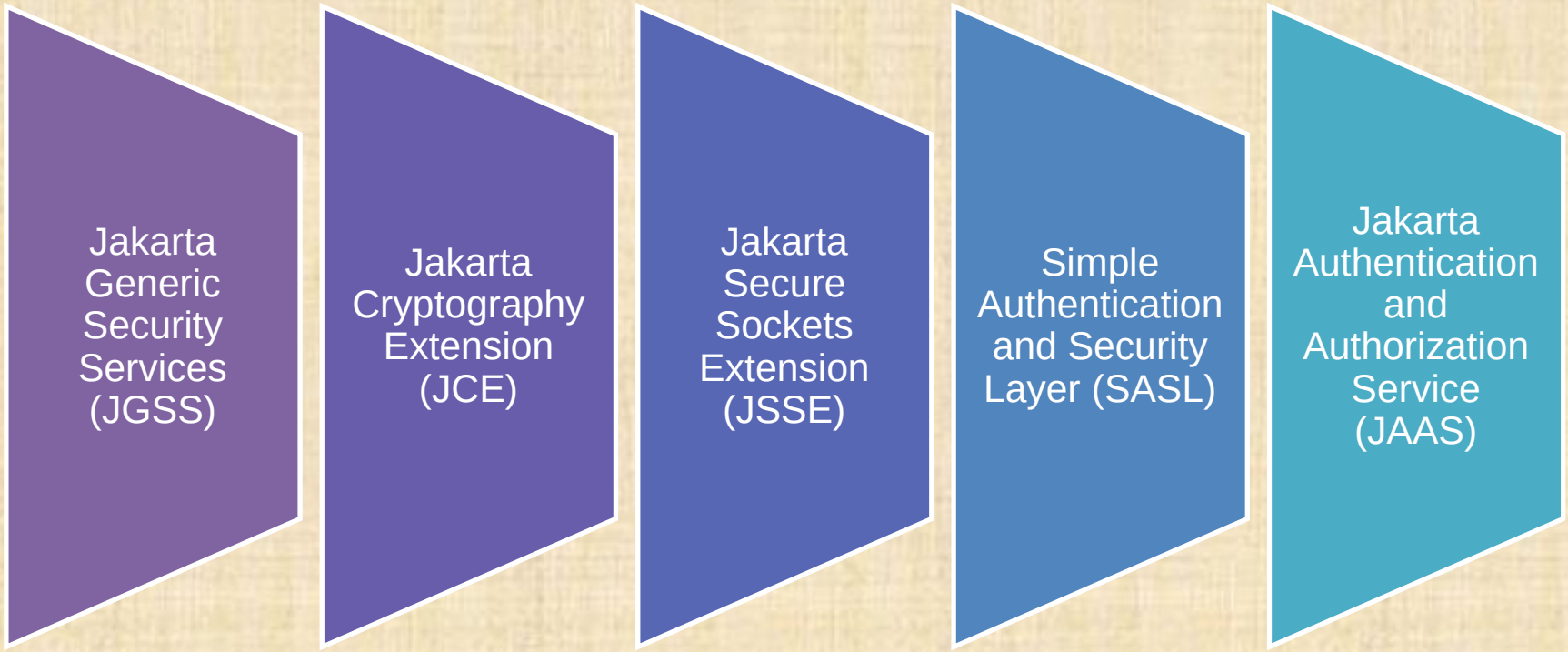
Security and Authentication 2-4

- Three levels in which application security can be implemented are:

- These three levels of security are in addition to the operating system level security provided to application data.

Security and Authentication 3-4

Security mechanisms provided by Jakarta EE:



Security and Authentication 4-4

- JAAS provides four classes and interfaces that are part of core class library of JAAS:

`LoginModule` `LoginContext` `Subject` `Principal`

They help in enabling independent development of authentication modules and the underlying authentication technologies.

- Applications enable authentication process by creating instances of `LoginContext` which in turn references `LoginModule` for performing authentication.

Basics of Using Application Roles, Security Constraints and Login Modules

- Three layers in the architecture used to create Enterprise applications are: **Presentation**, **Business**, and **Persistence**.
- These layers must interact with each other consistently. In Jakarta EE framework, CDI closes the gap between layers for reusable service components.
- A security constraint must include sub elements as shown in table:

Web resource collection:	**Authorization restriction:**
A list of HTTP operations and URL patterns that identify a set of resources the must be protected.	Names the roles qualified to execute the restricted requests and specifies whether authentication is to be used.

Summary

- CDI Bean is an application component that contains some business logic. Jakarta code or the unified EL can both use beans (expression language used in JSP and JSF technologies)
- Developers can specify the bean implementation version they want to use at runtime by using a Qualifier, which is a user-defined annotation.
- Interceptors allow the developers to interpose EJB business method. This means they allow the developer to add a wrapper code which is executed before or after the method is called.
- The CDI event notification facility employs essentially the same type safe methodology as the dependency injection service.
- The container is responsible for providing security services for the components deployed in it.

Session:14

Packaging Jakarta Applications

Jakarta Enterprise Beans in Action

Objectives

- Describe Packaging Enterprise Beans
- Explain Packaging Entities
- Explain Packaging Web Archives

Packaging Enterprise Beans

Different flavors of Java are differentiated based on the packages provided for the development of various kinds of applications.

Jakarta EE 9 makes use of package namespace `jakarta.` rather than `javax.` (which is still under control of Oracle).

An application that complies with the Jakarta EE platform before being installed is packaged into one or more standard units.

Each unit has a functional component or components, such as an applet, Web page, servlet, or Enterprise Bean and a deployment description with an optional content description.

Examples of Web Application Containing Jakarta Servlet

Shopping cart bean

Code Snippet 1:

```
package com.example.cart;

@Stateless
public class CartBean { ... }
```

Credit card processing bean

Code Snippet 2:

```
package com.example.cc;

@Stateless
public class CreditCardBean { ... }
```

Packaging and Deploying of a Jakarta EE Application

Application Component Providers create Jakarta EE modules, such as EJB, Web, Resource Adapter, and Application clients.

Application Assembler packages these modules to create a complete Jakarta EE Enterprise application.

Deployer deploys the deployable unit.

Types of Archive Files to Package the Modules

- `.jar` files
- `.war` files
- `.rar` files

Packaging Entities

- Entity classes are packaged as persistence units.
- A persistence unit refers to a set of logically connected entity classes and related configuration information on how the comprising entities are interrelated.
- Persistence units are defined in a configuration file `persistence.xml`. This descriptor file is added to the META-INF directory of the application.
- Code Snippet 3 shows the definition of a persistence unit in the `persistence.xml` file.

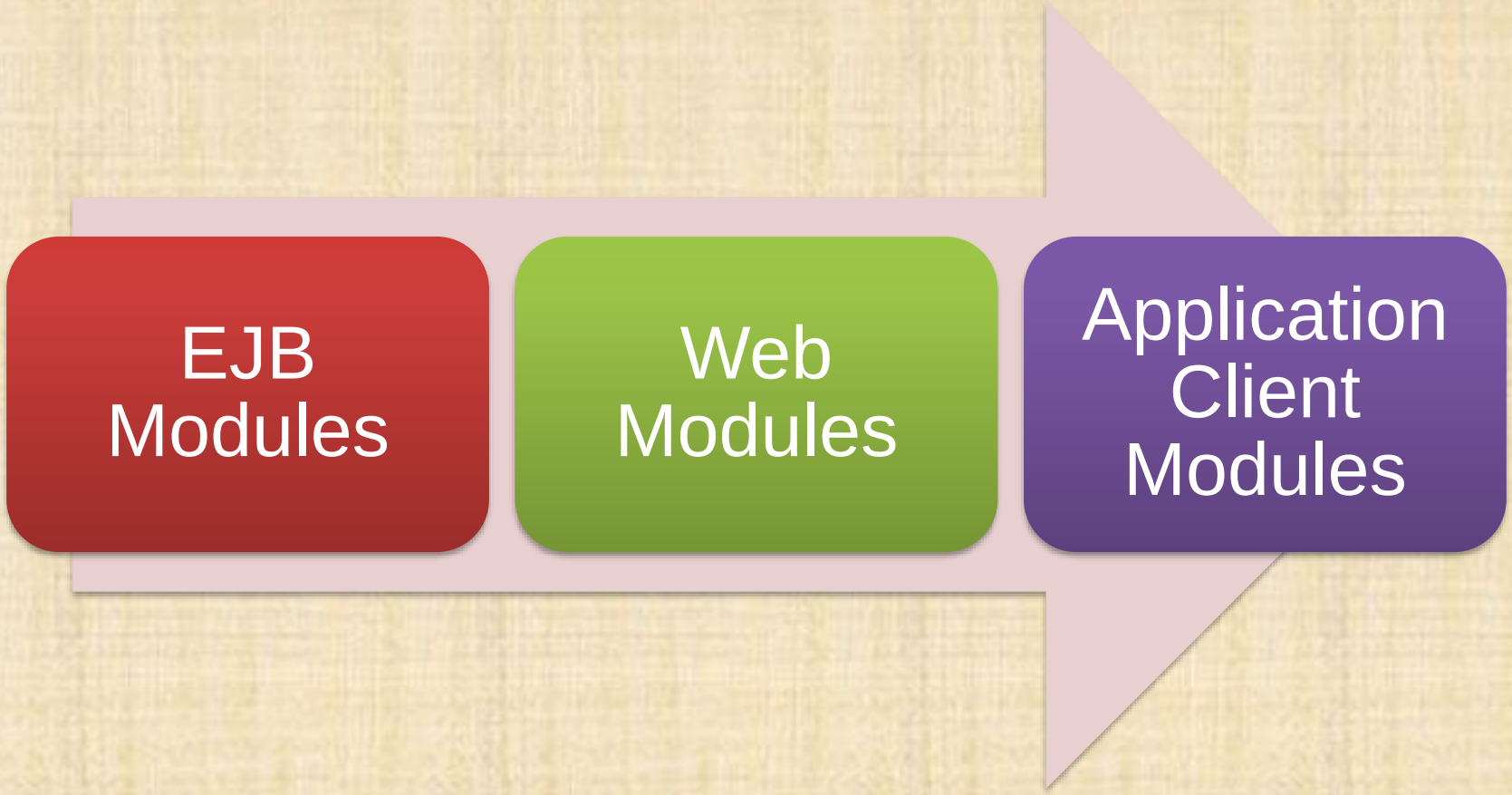
Code Snippet 3:

```
<persistence>
<persistence-unit name= "Operation">
<description>Project Description
</description>
<jta-data-source>jdbc/MyAccounts</jta-
data-source>
<jar-file>MyAccount.jar</jar-file>
<class>SavingsAccount</class>
<class>CurrentAccount</class>
</persistence-unit>
</persistence>
```

Descriptor File Elements

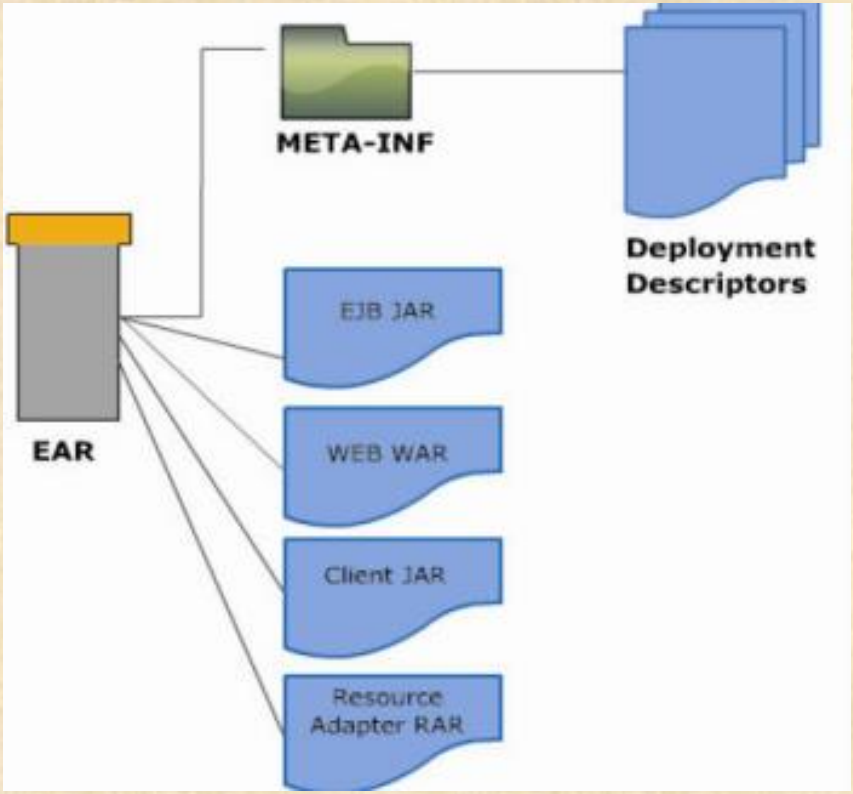
- <provider>
- <transaction-type>
- <mapping-file>

Jakarta EE Modules



Structure of EAR File

- A WAR or EAR file is a standard JAR (.jar) file with the extension .war or .ear.
- To manually package the components in an EAR file, the command can be run as: `jar cvf <name>.ear *.`



Summary

- One of the most interesting packages provided by Jakarta is Jakarta Enterprise Edition (Jakarta EE) platform that enables building of Enterprise applications.
- The Application Component Providers create Jakarta EE modules, such as EJB, Web, Resource Adapter, and Application clients. These modules can be deployed independently without being packaged into a Jakarta EE enterprise application.
- Entity classes are packaged as persistence units. A persistence unit refers to a set of logically connected entity classes and related configuration information on how the comprising entities are interrelated.
- A Jakarta EE application is distributed in the form of a Jakarta Archive (JAR), a Web Archive (WAR), or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard JAR (.jar) file with the extension .war or .ear.