Session: 4

Stateful Session Beans

Objectives

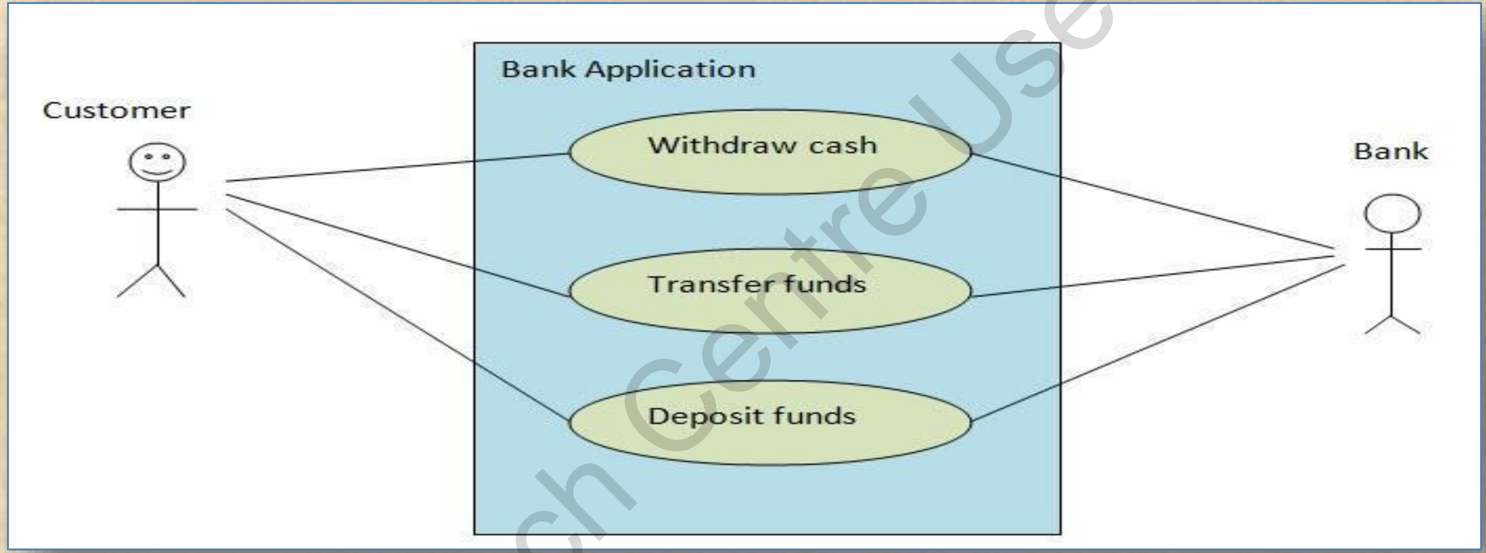
- ❑ Explain the working of Stateful Session beans
- ❑ Explain the different elements of Stateful Session beans
- ❑ Describe the lifecycle of Stateful Session beans and associated callback methods
- ❑ Explain the Implementation of Stateful Session beans
- ❑ Explain the different types of clients accessing Stateful Session bean
- ❑ Explain exception handling in session beans

Introduction 1-3

- Consider a situation where business processes invoked by the client needs the data to be maintained between the conversations over several requests.
- For example, while performing transactions on a bank account,
  - You may deposit as well as withdraw money from your account.

Introduction 2-3

- Following figure shows a typical use case of a Bank application where application state has to be maintained:



- All processes need separate method invocations in the application and the state of the account data has to be maintained in order to perform the transactions.

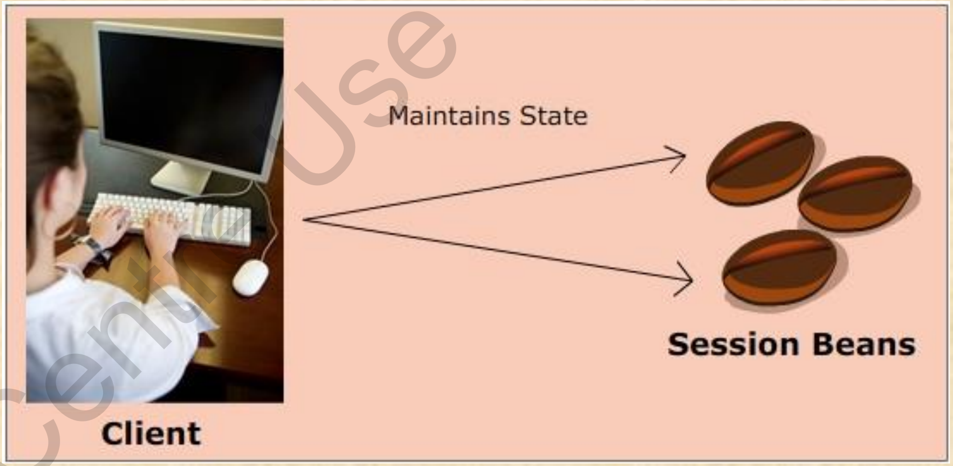
Introduction 3-3

- To process multiple requests, the enterprise bean specification has provided Stateful Session Beans.

- **Stateful Session Bean:**
  - Stateful Session bean can be defined as a bean that services business processes spanning over multiple business method requests.

Stateful Session Bean 1-2

- Stateful Session bean maintains the conversational state of the application client with which it is associated.

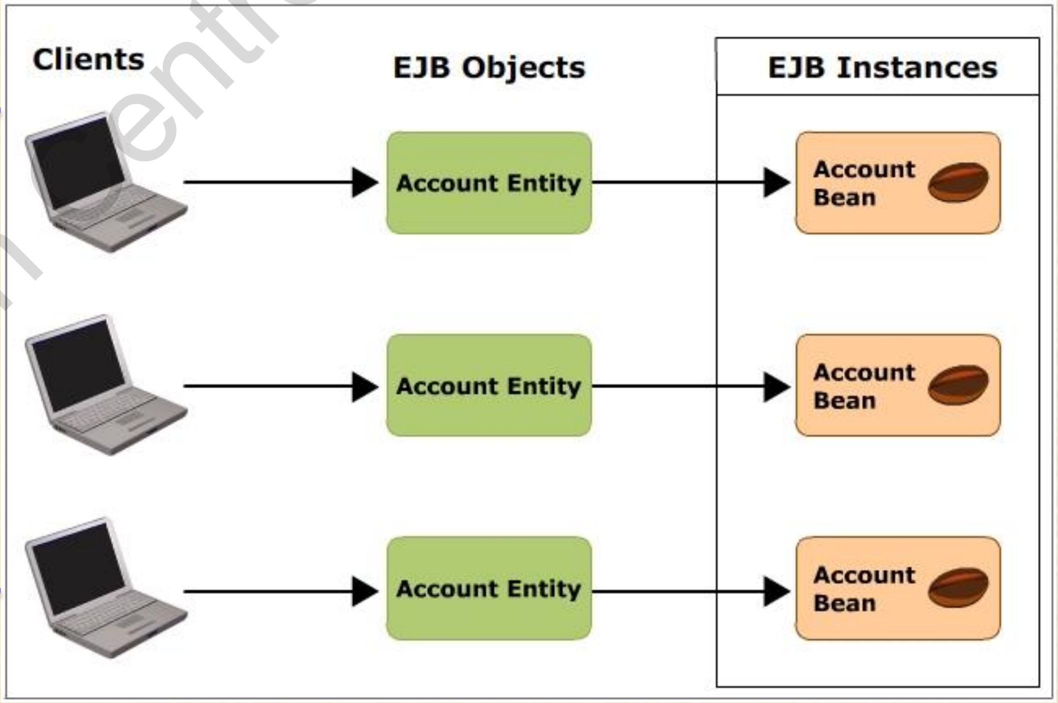


An example of a Stateful session bean is a shopping cart on an e-commerce Web site. Each time you add a product to the cart or go to the next Web page, a new request is performed while retaining the state of the previous requests.

Stateful Session Bean 2-2

- A Stateful Session bean is dedicated to a single client.
- It retains the state of the client, until the Stateful Session bean instance is explicitly removed by the container or there is a timeout.
- Following figure shows the implementation of Stateful Session beans in a Bank application:

Client can access the EJB instance of the Stateful Session Bean through the EJB object.



Characteristics of a Stateful Session Bean

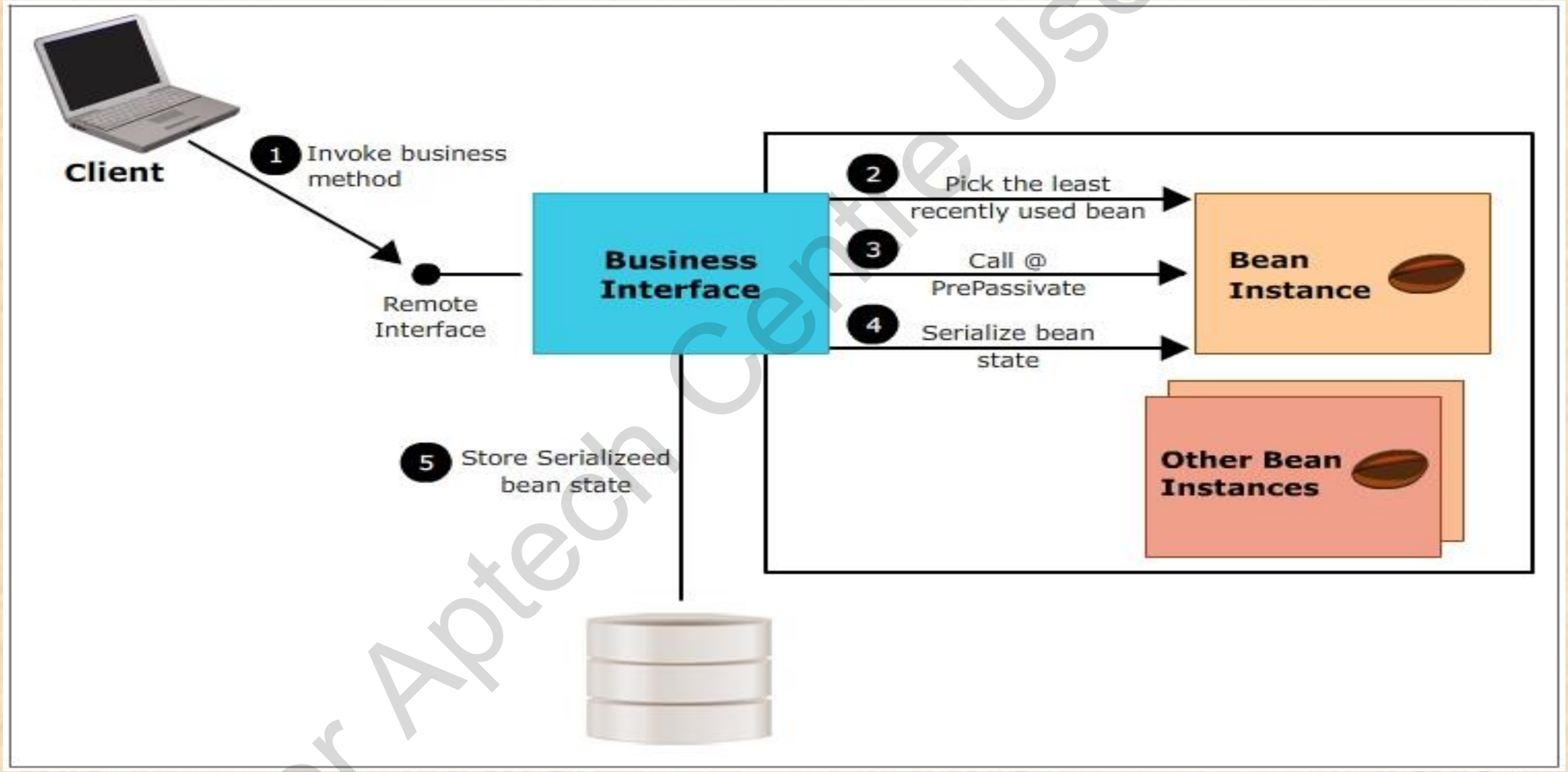
- ❑ Every instance of an application client is associated with a single instance of Stateful Session bean.
- ❑ Stateful Session beans can be activated or passivated.
- ❑ Stateful Session beans are transaction aware and short lived.
- ❑ Stateful Session beans are managed by EJB container.
- ❑ Stateful Session beans can access database, retrieve, and update data in the database.

Stateful Session Bean Conversational State 1-4

- When a Stateful Session bean is swapped out of the container its conversational state is written to the permanent storage.
- This process of writing the conversational state onto permanent storage and removing the Stateful Session bean from the container is known as passivation.

Stateful Session Bean Conversational State 2-4

- Following figure shows the process of storing the conversational state of a stateful session bean:

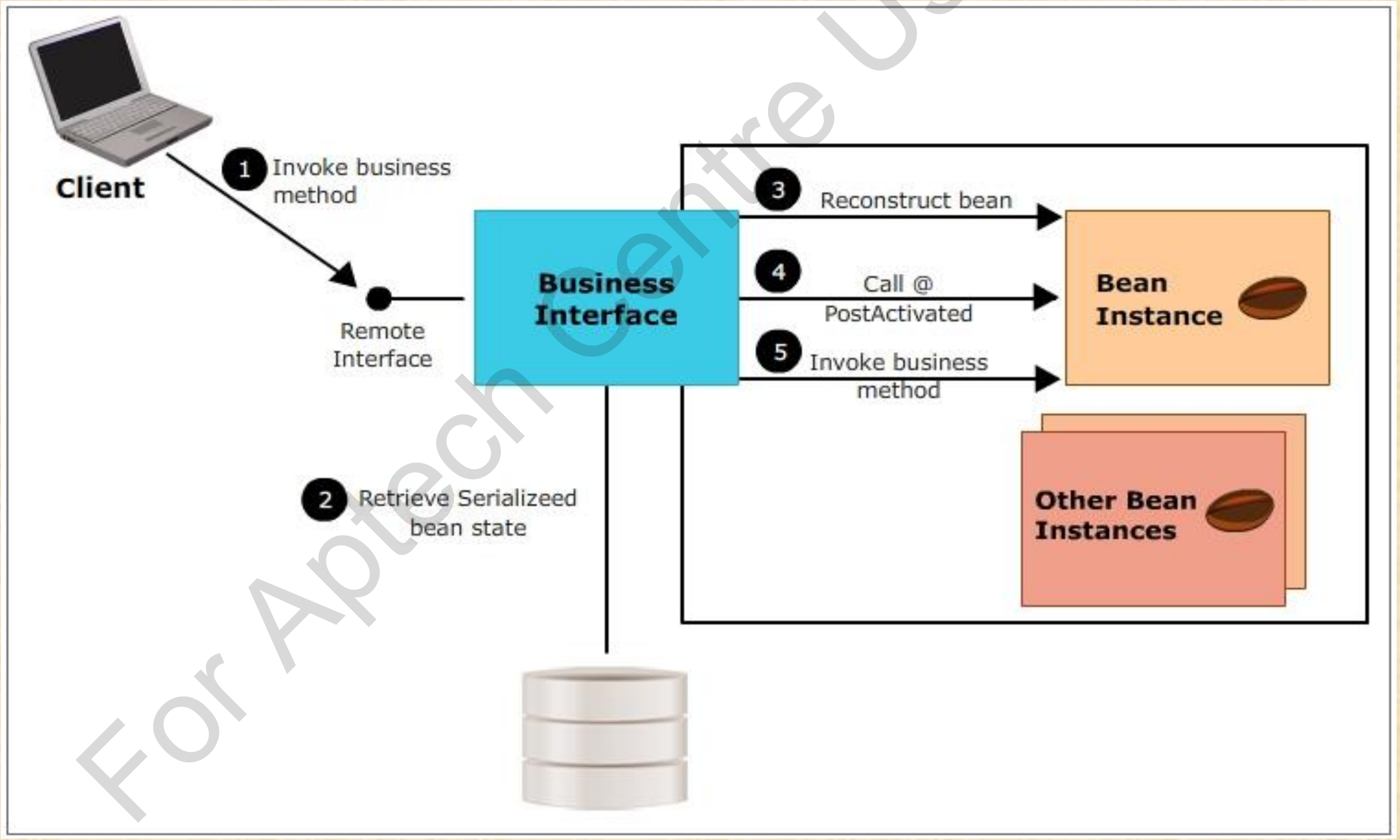


Stateful Session Bean Conversational State 3-4

- To choose which Stateful Session Bean must be removed from the container, the container generally uses Least Recently Used (LRU) strategy.
- When there is a request for the swapped out bean, then it is again brought back into the container, this process of bringing back the bean into container is known as activation.

Stateful Session Bean Conversational State 4-4

- Following figure shows the process of retrieving the conversational state of a Stateful Session bean:



Elements of Stateful Session Bean 1-2

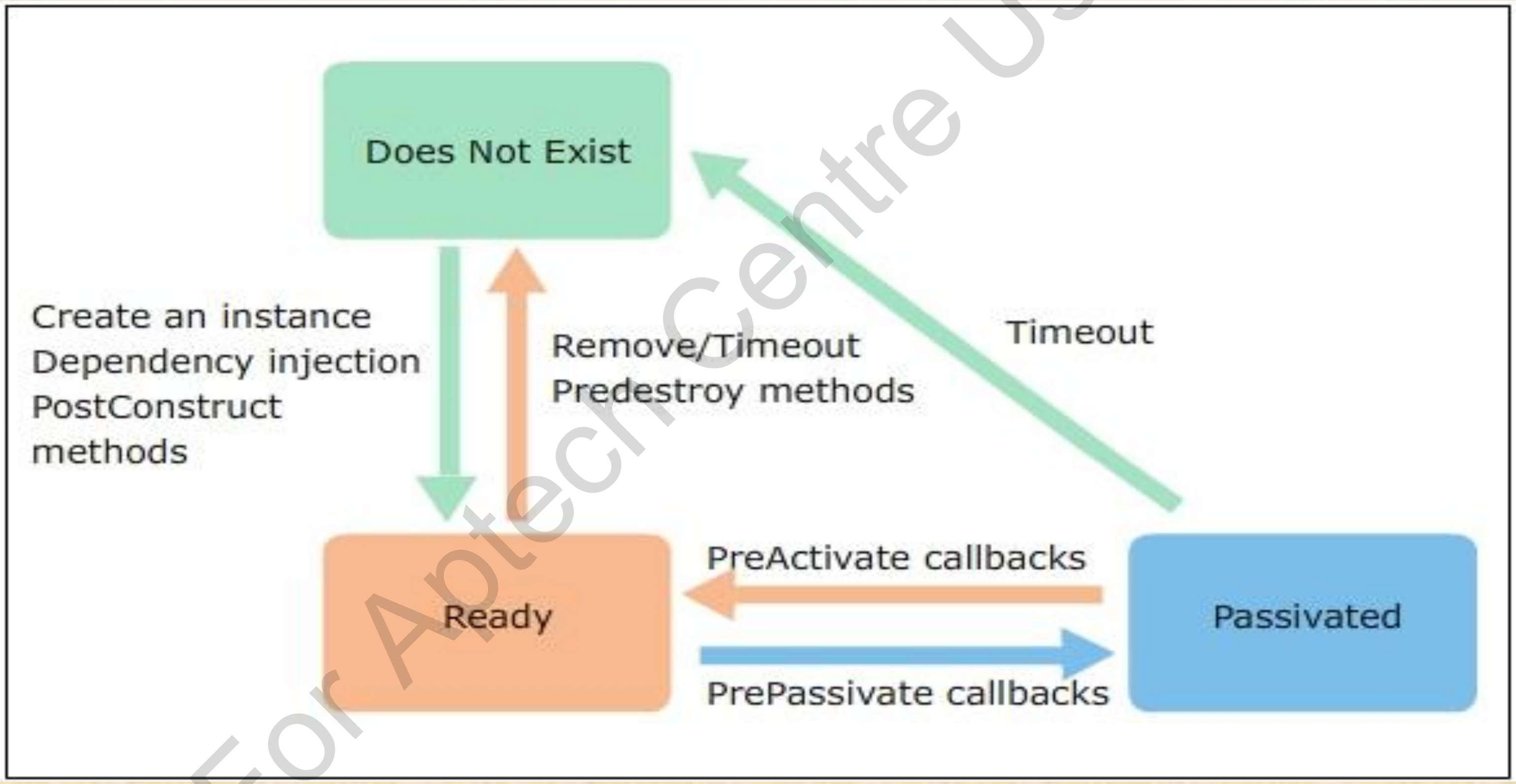
Bean class	• Is a simple Java class that uses a class level annotation or deployment descriptor to specify the bean type. • Annotated using a class level `@Stateful` annotation.
Business interface	• Defines the functions to be accessed by clients. • Annotated using `@local` or `@remote` interface.
Business methods	• Implement the functionality of the bean class.

Elements of Stateful Session Bean 2-2

- Since, SOAP-based Web services are Stateless, Stateful Session Bean cannot have a Web service endpoint interface.

Life Cycle of a Stateful Session Bean 1-4

- Following figure shows different stages in a Stateful Session bean:



Life Cycle of a Stateful Session Bean 2-4

Following are the steps performed by the container during the life cycle:

- Uses the default constructor to create a bean instance.
- Injects the resources such as database connections.
- Stores the bean instance in the memory.
- Executes the business method invoked by the client.
- Waits and executes further requests.
- Passivates the bean instance when the client is idle.
- Activates the bean instance on receiving a call from the client.
- Destroys a bean instance when the bean is not invoked for a period of time.
- Requests for removal of bean instance from the client requiring the activation of the bean instance followed by destruction.

Life Cycle of a Stateful Session Bean 3-4

In **Does Not Exist State**, a Stateful Session bean has not been instantiated by the container.

Once the bean is instantiated the container allocates required resources through dependency injection.

In **Ready state**, the session bean can accept and service client requests.

A session bean is passivated when the container has to persist session data onto secondary storage.

Bean may transit from **Ready state** to **Does Not Exist** state when the container removes it.

Life Cycle of a Stateful Session Bean 4-4

Passivated State

- During states of inactivity container may transform a session bean to passive state.
- The container stores the conversational state and places the session bean in a passivated state.
- Developers can define PrePassivate and PostActivate callback methods.
- Every Stateful Session bean is associated with a maximum timeout period for which it can stay in passivated state.
- When the container receives a request from the client, the container transits the session bean from **Passivated State** to **Ready State**.

Life Cycle Callback Methods

PostConstruct	• Methods to be executed after the session bean is instantiated by the container. • Are prefixed with `@PostConstruct` annotation.
PreDestroy	• Methods to be executed before the bean is removed from the container and transits to **Does not Exist** state. • Are prefixed with `@PreDestroy` annotation.
Prepassivate	• Methods to be executed before the session bean transits from Ready state to Passivated state. • Annotated with `@Prepassivate` annotation.
PostActivate	• Methods to be executed when the bean transits from the passivated state to Ready state. • Annotated with `@PostActivate` annotation.

Programming Rules for Stateful Session Bean

- Following are the rules to be implemented by a Stateful Session beans:
  - Instances of Stateful Session bean should be of Java primitive data types or Serializable objects.
  - Stateful Session bean class should define a method that would destroy the bean instance by the bean class using the `@Remove` annotation.
  - Stateful Session bean have the PostActivate and PrePassivate lifecycle callback methods.
  - The **`PostActivate`** method is invoked once the bean is brought back in the memory.
  - The **`PrePassivate`** method is invoked before the bean instance is passivated.

Developing Stateful Session Bean 1-2

- Following code snippet demonstrates creating **ProductCatalogBean** class:

```
. . .
@Stateful
public class ProductCatalogBean implements ProductCatalogBeanRemote {

   List<String> products;
   public ProductCatalogBean(){
      products = new ArrayList<String>();
   }
    public void addProduct(String productName) {
      products.add(productName);
   }

   public List<String> getProducts() {
      return products;
   }
}
```

- Two business methods `addProduct()` and `getProducts()` are being created.

Developing Stateful Session Bean 2-2

- Following code snippet shows the remote interface of a **`ProductCatalogBean`** class:

```
. . .
@Remote
public interface ProductCatalogBeanRemote {
  void addProduct(String productName);
   List getProducts();
}
```

  - The annotation `@Remote` indicates that these methods can be accessed only by the remote clients.

Clients Interfaces for Session Beans 1-2

An EJB by itself cannot perform any function till it is not invoked by a client.

The client invokes bean operations through interfaces.

A client can access the application through local interface, remote interface, or Web service.

Clients Interfaces for Session Beans 2-2

❑Following are the factors which influence the decision of whether the clients should access the bean locally or remotely:

- Type of Service
- Relation among the bean components
- Location of components on the enterprise network
- Performance demands

Clients of Enterprise Applications

- Following are different types of clients for enterprise applications:

Local clients

Remote clients

Web clients

Local Client

- A local client and the session bean are located on the same JVM.
- A local client can be another enterprise bean or Web component of the application.
- Local clients can access the session beans either through no-interface or through local interface view.

Remote Client 1-4

- Remote client may reside on a different JVM or a different physical machine.
- A remote client can be another enterprise bean residing in the same or different location.
- A remote client can also be a Web application, an applet, or a Java console application.
- Remote clients cannot access enterprise beans through no-interface view.
- Remote clients has to implement a business interface which is annotated with `@Remote` annotation.

Remote Client 2-4

❑ Following code snippet shows a client accessing a shopping portal application:

```
. . .
public class StatefulClient{
 @EJB
 private static ProductCatalogBeanRemote
productCatalogBean;
 public static void main(String[] args) {
    List PList = new ArrayList();
    productCatalogBean.addProduct("Laptop");
    productCatalogBean.addProduct("MobilePhone");
    productCatalogBean.addProduct("Personal Digital
Assistant");
```

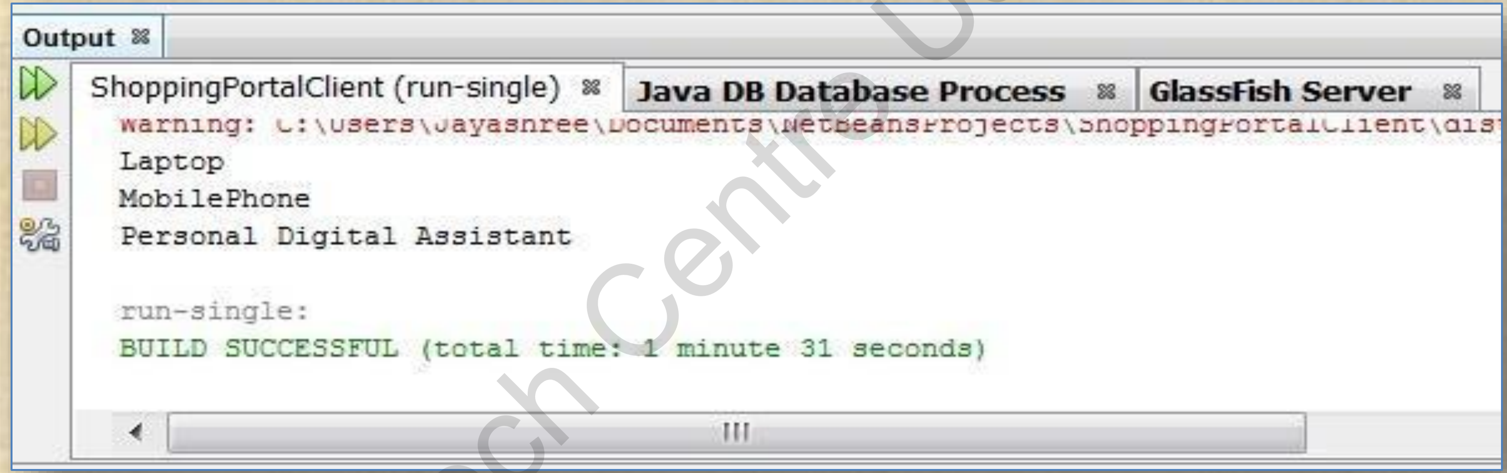
Remote Client 3-4

```
      productCatalogBean.getProducts();
     // Iterate through all the elements of the collection
     Iterator itr = PList.iterator();
      while (itr.hasNext()) {
           String str = (String) itr.next();
           System.out.print(str + "\n");
       }
       System.out.println();
     }

}
```

Remote Client 4-4

- Following figure shows the output of executing a remote client of the **`ShoppingPortal`** application:



```
warning: C:\Users\Jayashree\Documents\NetBeansProjects\ShoppingPortalClient\dis
Laptop
MobilePhone
Personal Digital Assistant

run-single:
BUILD SUCCESSFUL (total time: 1 minute 31 seconds)
```

Accessing Through JNDI Lookups 1-3

- The client can access the local or remote interface of an enterprise bean either through EJB objects or through lookup services such as JNDI.
- JNDI is a naming and directory service provided by Java platform.

Accessing Through JNDI Lookups 2-3

- Following are the components which need to be set to access interfaces through JNDI lookup:

JNDI initialization parameters

- Includes configuring a JNDI driver.
- Client must provide required properties to access the JNDI driver.
- InitialContext provided by the JNDI API is used to set container specific properties.

InitialContext class

- Used to create an entry point into the naming system.
- InitialContext provides the root to the hierarchy of the JNDI names in an application.

Accessing Through JNDI Lookups 3-3

- Following code snippet demonstrates the usage of **`InitialContext`** object:

```
. . .
Context c = new InitialContext();
return (ProductCatalogBeanRemote)
c.lookup("java:global/ShoppingPortal/ShoppingPortal-
ejb/ProductCatalogBean!beans.ProductCatalogBeanRemote"
);

 . . .
```

- The code shows the usage of `InitialContext` object which returns the reference of `ProductCataloBeanRemote` interface.

Configuring JNDI 1-2

- A lookup operation in the JNDI namespace is performed through **`lookup()`** method.
- The **`lookup()`** method uses the deployment descriptor to obtain the JNDI name and object binding.

Configuring JNDI 2-2

- Following code snippet shows a sample deployment descriptor with JNDI name bindings defined:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD
GlassFish Application Server 3.1 EJB 3.1//EN"
"http://glassfish.org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>ProductCatalogBean</ejb-name>
      <jndi-name>pro</jndi-name>
    </ejb>
  </enterprise-beans>
</glassfish-ejb-jar>
```

- In Code Snippet, the `ProductCatalogBean` is bound to the JNDI name ‘`pro`’.

Injecting Resources into Enterprise Beans

- Resource injection enables developers to inject resources such as databases, connectors, and so on into container-managed components.
- Resources to be injected must be defined in the JNDI namespace.
- Resources are referred through **`@javax.annotation.Resource`** annotation.
- **`@Resource`** annotation has six attributes – **`name, type, authenticationType, shareable, mappedName`**, and **`description`**.
- Resources can be injected at the following levels in the code:
  - Field level
  - Method level
  - Class level

Exception Handling in Session Beans

- Exception is an unusual condition/control flow in an application.
- Exceptions might be caused by user errors or programming errors.
- Exception handling is done through:
  - **`try-catch`** block
  - **`throws`** statement

Types of Exceptions

❑ There are three categories of Exceptions:

Checked Exceptions

- Occurs due to invalid conditions
- Outside immediate control of program

Unchecked Exceptions

- Occurs due to error in programming logic
- Cannot be detected at compile time

Runtime Exceptions

- Occurs due to incorrect logic
- Detected only when the application executes

Java EE Exceptions 1-2

- There are two types of Java EE exceptions:

Application exceptions

- Arise due to certain conditions in the business logic of the application.
- Example: `CreateException`, `FinderException`, and `RemoveException`.

System exceptions

- Arise due to faults in the application infrastructure or failure of database connections.
- Example: `RemoteException`, `NamingException`, and `SQLException`.

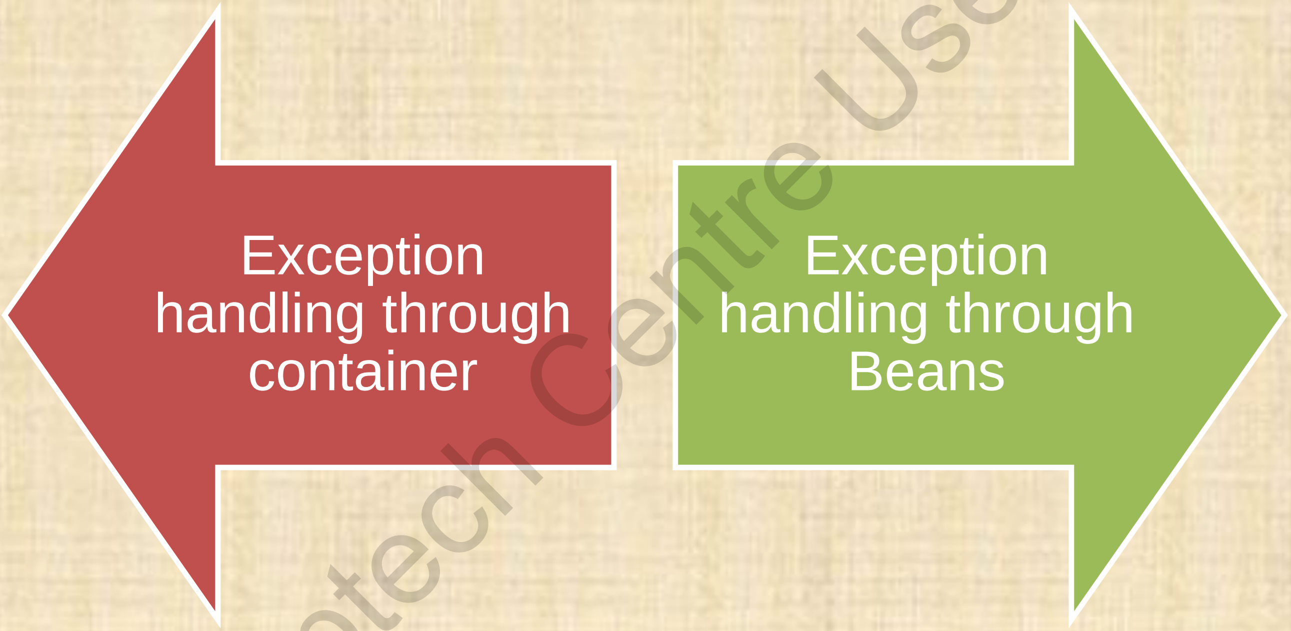
Java EE Exceptions 2-2

- Following table shows different exceptions that might occur in EJB applications:

Exception	Description
`CreateException`	Raised when the instantiation of an object or bean fails.
`FinderException`	Raised when the application is unable to find an object it is looking up for.
`RemoveException`	Raised when the application cannot remove an instance of bean, cannot be removed from the container.
`RemoteException`	Raised when there is a network failure and the client cannot access a remote object.
`NamingException`	Raised when the jndi name could not be resolved to the object which is looked up for.
`SQLException`	Raised when the bean could not get response from the application database.
`EJBException`	Raised when the bean could not respond to the application clients appropriately.

Exception Handling in Enterprise Applications

Exceptions in enterprise applications are handled in two ways:


Exception handling through container
Exception handling through Beans

Handling Exceptions Through Containers

Container handles application exceptions by returning it to caller and executing the exception handling code.

Container performs the following operations to perform system exceptions:

- Container logs the system exception
- Deallocates the allocated resources and performs clean up operation
- Removes bean instance from the memory
- The calling method is informed about the exception thrown

Handling Exceptions Through Beans 1-3

- Beans handle exceptions by writing explicit code.
- Following code snippet demonstrates usage of exceptions in enterprise applications:

```
. . .
public class ExceptionDemo {
    public static void main(String[] args) throws
FileNotFoundException, IOException {
        try{
            testException(-5);
            testException(-10);
        }catch(FileNotFoundException e){
            e.printStackTrace();
       }catch(IOException e){
            e.printStackTrace();
        }finally{
            System.out.println("Releasing resources");
        }
        testException(15);
    }
```

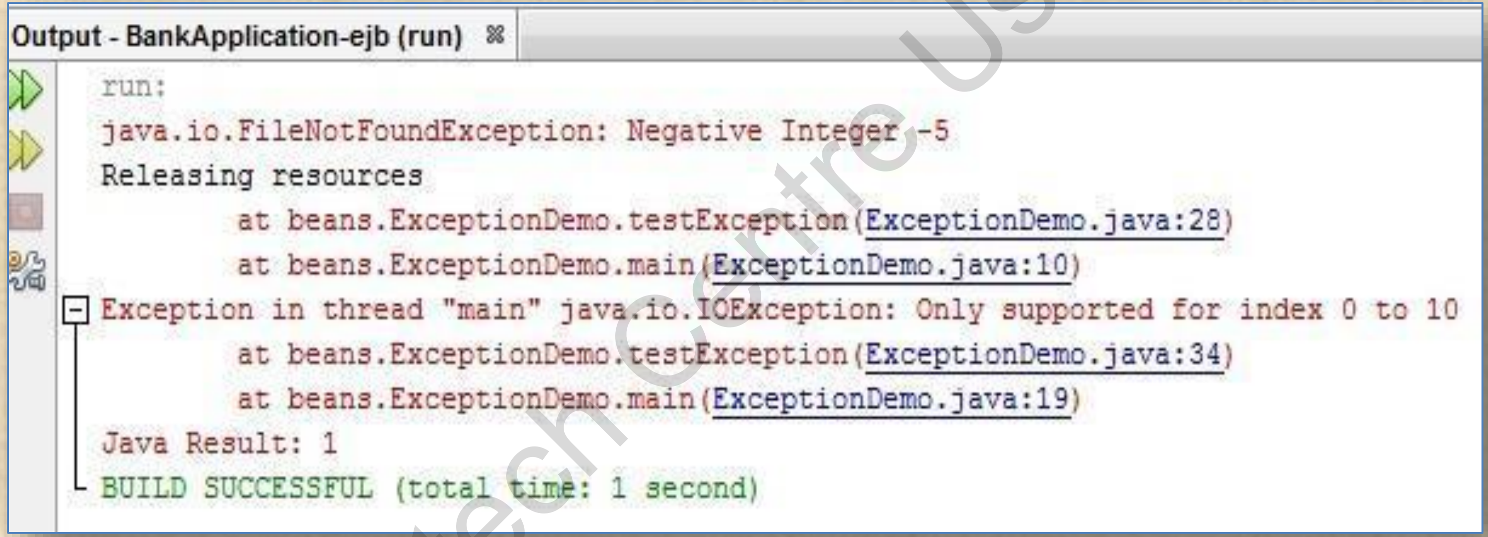
Handling Exceptions Through Beans 2-3

```
public static void testException(int i) throws 
FileNotFoundException, IOException{

        if(i < 0){
            FileNotFoundException myException = new 
FileNotFoundException("Negative Integer "+i);
            throw myException;
        }else if(i > 10){
            throw new IOException("Only supported for index 0 
to 10");
        }
    }
}
```

Handling Exceptions Through Beans 3-3

- Following figure shows the output of the exception handling code:


Output - BankApplication-ejb (run)

```
run:
java.io.FileNotFoundException: Negative Integer -5
Releasing resources
        at beans.ExceptionDemo.testException(ExceptionDemo.java:28)
        at beans.ExceptionDemo.main(ExceptionDemo.java:10)
Exception in thread "main" java.io.IOException: Only supported for index 0 to 10
        at beans.ExceptionDemo.testException(ExceptionDemo.java:34)
        at beans.ExceptionDemo.main(ExceptionDemo.java:19)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

Exception Logging 1-4

- Exceptions are logged by applications for further analysis.
- Java provides **`java.util.logging`** package to implement logging.
- Applications implement logging to:
  - Diagnose any problem in the application
  - Trace the application functionality
  - Root cause the problem in case of a system crash

Exception Logging 2-4

- Following code snippet demonstrates exception logging in enterprise applications:

```
private com.card.CardValidationRemote
lookupCardValidationBean() {
try {
javax.naming.Context c = new javax.naming.InitialContext();
Object remote = c.lookup(...);
com.card.CardValidationRemoteHome rv = ... return
rv.create();
} catch(javax.naming.NamingException ne) {

java.util.logging.Logger.getLogger(getClass().getName()).lo
g java.util.logging.Level.SEVERE,"exception caught" ,ne);
      throw new RuntimeException(ne);
```

Exception Logging 3-4

```
        } catch(javax.ejb.CreateException ce) {
        java.util.logging.Logger.getLogger(getClass().getName
()).log(java.util.logging.Level.SEVERE,"exception caught"
,ce);
        throw new RuntimeException(ce);
        } catch(java.rmi.RemoteException re) {

java.util.logging.Logger.getLogger(getClass().getName()).log
(java.util.logging.Level.SEVERE,"exception caught" ,re);
        throw new RuntimeException(re);
       }
}
```

Exception Logging 4-4

`java.util.logging` package provides various levels of logging in the application.

Following are predefined level constants:

- `SEVERE (highest value)`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST (lowest value)`

Summary

- Stateful Session beans store the conversational state of the session.
- Each Stateful Session bean has a unique identity and is associated with a single client.
- There are three states in the life cycle of a Stateful Session bean – Does not Exist, Activated, and Passivated.
- There are four categories of life cycle callback methods – PostConstruct, PrePassivate, PostActivate, and PreDestroy.
- Stateful Session beans can be accessed through both local and remote interface.
- Stateful Session beans can be activated through local, remote, and Web service clients.
- Application and system exceptions are two types of exceptions in an enterprise application according to EJB specification.