CMR
INSTITUTE OF
TECHNOLOGY

										USN

Internal Assessment Test 3– May 2017

Sub:	Advanced Java Programming							Code:	13MCA 42
Date:	30.05.2017	*Duration:*	*90 mins*	*Max Marks:*	*50*	*Sem:*	*4*	*Branch:*	*MCA*

Answer any five of the following. **5x10=50M**

1.Explain the various steps of JDBC with code snippet

Seven Basic Steps in Using JDBC

1. Load the Driver
2. Define the Connection UR
3. Establish the Connection
4. Create a Statement Object
5. Execute a query
6. Process the results
7. Close the Connection

1. Load the JDBC driver

When a driver class is first loaded, it registers itself with the driver Manager Therefore, to register a driver, just load it!

Example:

String driver = "sun.jdbc.odbc.JdbcOdbcDriver"; Class.forName(driver);

Or

Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);

2. Define the Connection URL

jdbc : subprotocol : source

- each driver has its own subprotocol

each subprotocol has its own syntax for the source

jdbc : odbc : DataSource

Ex: jdbc : odbc : Employee

jdbc:msql://host[:port]/database

Ex: jdbc:msql://foo.nowhere.com:4333/accounting

3. Establish the Connection

- DriverManager Connects to given JDBC URL with given **user name** and **password**
- **Throws** java.sql.SQLException
- **returns** a Connection object
- A Connection represents a session with a specific database.
- The connection to the database is established by **getConnection()**, which requests access to the database from the DBMS.
- A Connection object is returned by the getConnection() if access is granted; else getConnection() throws a SQLException.
- If username & password is required then those information need to be supplied to access the database.

String url = jdbc : odbc : Employee;

Connection c = DriverManager.getConnection(url,userID,password);

- Sometimes a DBMS requires extra information besides userID & password to grant access to the database.
- This additional information is referred as properties and must be associated with Properties or Sometimes DBMS grants access to a database to anyone without using username or password.

Ex: Connection c = DriverManager.getConnection(url) ;

4. Create a Statement Object

A Statement object is used for executing a static SQL statement and obtaining the results produced by it.

Statement stmt = con.createStatement();

This statement creates a Statement object, *stmt* that can pass SQL statements to the DBMS using connection, *con*.

5. Execute a query

Execute a SQL query such as SELECT, INSERT, DELETE, UPDATE Example

String SelectStudent= "select * from STUDENT";

6. Process the results

- A ResultSet provides access to a table of data generated by executing a Statement.
- Only one ResultSet per Statement can be open at once.
- The table rows are retrieved in sequence.
- A ResultSet maintains a cursor pointing to its current row of data.
- The 'next' method moves the cursor to the next row.

7. Close the Connection

connection.close();

- Since opening a connection is expensive, postpone this step if additional database operations are expected

2.**Briefly explain the following**

i)packages

ii)interfaces

An interface in java is a blueprint of a class. It has static constants and abstract methods only. The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also represents IS-A relationship.

It cannot be instantiated just like abstract class.

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

The interface keyword is used to declare an interface. Here is a simple example to declare an interface:

```
/* File name : NameOfInterface.java */

Import java.lang.*;

//Any number of import statements
```

Public interface NameOfInterface

{

//Any number of final, static fields

//Any number of abstract method declarations\

}

Ex:

```
/* File name : Animal.java */

interfaceAnimal{

publicvoid eat();

publicvoid travel();

}
```

Implementing Interfaces:

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
classclassname[extendssuperclass] [implementsinterface[,interface...]] {

// class-body

}
```

Ex:

```
/* File name : MammalInt.java */

Public class MammalInt implements Animal{

publicvoid eat(){

System.out.println("Mammal eats");

}

Public void travel(){

System.out.println("Mammal travels");

}

Public int noOfLegs(){

return 0;

}
```

```
Public static void main(Stringargs[]){

MammalInt m =new MammalInt();

m.eat();

m.travel();

}

}
```

Package:

A java package is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Ex:

```
package mypack;
 public class Simple{
 public static void main(String args[]){
System.out.println("Welcome to package");
}
}
```

3.Explain built in annotations in detail

Java Annotations

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.
First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Built-In Java Annotations

There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.
Built-In Java Annotations used in java code
@Override
@SuppressWarnings
@Deprecated
Built-In Java Annotations used in other annotations
@Target
@Retention
@Inherited
@Documented

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.
Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

```
class Animal{
void eatSomething(){System.out.println("eating something");}
}

class Dog extends Animal{
@Override
void eatsomething(){System.out.println("eating foods");}//should be eatSometh
ing
}

class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

Output:

Comple Time Error

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;
class TestAnnotation2{
@SuppressWarnings("unchecked")
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("sonoo");
list.add("vimal");
list.add("ratan");

for(Object obj:list)
System.out.println(obj);

}}
```

Now no warning at compile time.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
class A{
void m(){System.out.println("hello m");}

@Deprecated
void n(){System.out.println("hello n");}
}

class TestAnnotation3{
public static void main(String args[]){

A a=new A();
a.n();
```

}}
At Compile Time:
Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.
At Runtime:
hello n
@Target

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.

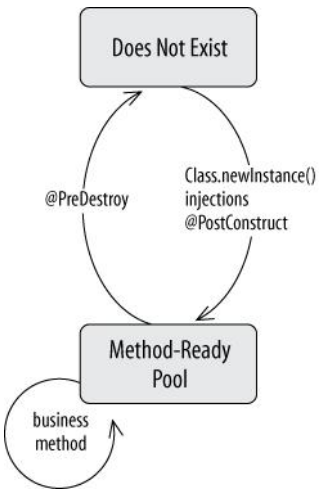
Let's see the constants of ElementTypeenum:

Element Types Where the annotation can be applied

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

4.(a) Explain the lifecycle of MDB

As session beans have well-defined lifecycles, so does the message-driven bean. The MDB instance's lifecycle has two states: *Does Not Exist* and *Method-Ready Pool*.



The Does Not Exist State

When an MDB instance is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready Pool

MDB instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it may create a number of MDB instances and enter them into the Method-Ready Pool (the actual behavior of the server depends on the implementation).

When the number of MDB instances handling incoming messages is insufficient, more can be created and added to the pool.

Transitioning to the Method-Ready Pool

When an instance transitions from the Does Not Exist state to the Method-Ready Pool, three operations are performed on it. First, the bean instance is instantiated by invoking the Class.newInstance() method on the bean implementation class. Second, the container injects any resources that the bean's metadata has requested via an injection annotation or XML deployment descriptor.

Finally, the EJB container will invoke the PostConstruct callback if there is one.The bean class may or may not have a method that is annotatedwith @javax.ejb.PostConstruct. If it is present, the container will call this annotated method after the bean is instantiated. This @PostConstruct annotated method can be of any name and visibility, but it must return void, have no parameters, and throw nochecked exceptions. The bean class may define only one @PostConstruct method (but it is not required to do so).

```
@MessageDriven
public class MyBean implements MessageListener {
  @PostConstruct
  public void myInit() {}
```

MDBs are not subject to activation, so they can maintain open connections to resources for their entire lifecycles.

The @PreDestroy method should close any open resources before the stateless session bean is evicted from memory at the end of its lifecycle.

Life in the Method-Ready Pool

When a message is delivered to an MDB, it is delegated to any available instance in the Method-Ready Pool. While the instance is executing the request, it is unavailable to process other messages. The MDB can handle many messages simultaneously, delegatingthe responsibility of handling each message to a different MDB instance. When a message is delegated to an instance by the container, the MDB instance's Message Driven Context changes to reflect the new transaction context. Once the instance has finished, it is immediately available to handle a new message.

Transitioning out of the Method-Ready Pool: The death of an MDB instance

Bean instances leave the Method-Ready Pool for the Does Not Exist state when the server no longer needs them—that is, when the server decides to reduce the total size of the Method-Ready Pool by evicting one or more instances from memory. The process begins by invoking an @PreDestroy callback method on the bean instance. Again, as with @PostConstruct, this callback method is optional to implement and its signature must return a void type, have zero parameters, and throw no checked exceptions. A @PreDestroy callback method can perform any cleanup operation, such as closing open resources.

```
@MessageDriven
public class MyBean implements MessageListener {
  @PreDestroy
```

public void cleanup( ) {

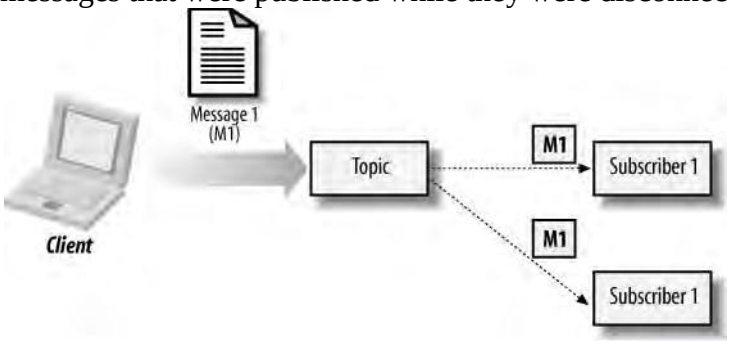
(b) Explain the JMS Messaging models with neat diagram

JMS provides two types of messaging models: *publish-and-subscribe* and *point-topoint*. The JMS specification refers to these as *messaging domains*. In JMS terminology, publish-and-subscribe and point-to-point are frequently shortened to *pub/sub* and *p2p* (or *PTP*), respectively.

Publish-and-subscribe

In publish-and-subscribe messaging, one producer can send a message to many consumers through a virtual channel called a topic. Consumers can choose to subscribe to a topic. Any messages addressed to a topic are delivered to all the topic's consumers.
The pub/sub messaging model is largely a push-based model, in which messages are automatically broadcast to consumers without the consumers having to request or poll the topic for new messages.

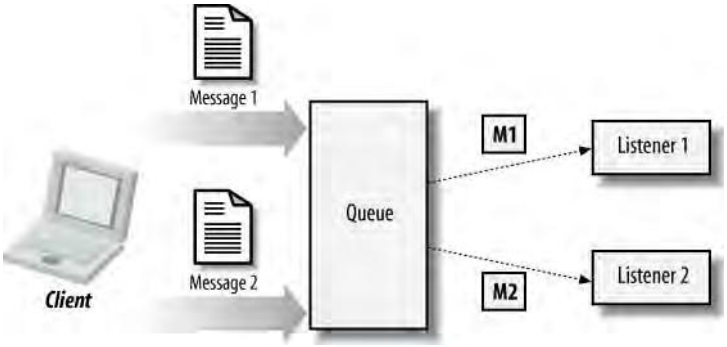
In this pub/sub messaging model, the producer sending the message is not dependent on the consumers receiving the message. JMS clients that use pub/sub can establish durable subscriptions that allow consumers to disconnect and later reconnect and collect messages that were published while they were disconnected.



Point-to-point

The point-to-point messaging model allows JMS clients to send and receive messages both synchronously and asynchronously via virtual channels known as *queues*. The p2p messaging model has traditionally been a *pull-* or *polling-based model*, in which messages are requested from the queue instead of being pushed to the client automatically.

A queue may have multiple receivers, but only one receiver may receive each message. As shown earlier, the JMS provider takes care of doling out the messages among JMS clients, ensuring that each message is processed by only one consumer. The JMS specification does not dictate the rules for distributing messages among multiple receivers.



5. What is entity bean Explain the JAVA persistence model in detail

• An entity bean is a remote object that
– manages persistent data,
– performs complex business logic,
– uses several dependent Java objects, and
– can be uniquely identified by a primary key.
• Entity beans are normally coarse-grained persistent objects,
They utilize persistent data stored within several fine-grained persistent Java objects.
Persistence
• JPA (Java Persistent API) - handles the plumbing between Java and SQL. EJB provides convenient integration with JPA via the entity bean.
• **Persistence is a key piece of the Java EE platform**.
– Older versionJ2EE, the EJB 2.x specification was responsible for defining this layer.
– In Java EE 5, persistence was spun off into its own specification.
– EE6, we have a new revision called the Java Persistence API, Version 2.0, or JPA.
• **Persistence**
– **provides an ease-of-use** abstraction on top of JDBC
– so that your code may be isolated from the database
– and vendor-specific peculiarities and optimizations.
• It can also be described as an object-to-relational mapping engine (ORM), which means that the Java Persistence API can automatically map your Java objects to and from a relational database.
• **Entity** - A persistent object representing the data-store record. It is good to be serializable.
• **EntityManager** - Persistence interface to do data operations like add/delete/update/find on persistent object(entity). It also helps to execute queries using **Query** interface.
• **Persistence unit (persistence.xml)** - Persistence unit describes the properties of persistence mechanism.
• **Data Source (*ds.xml)** - Data Source describes the data-store related properties like connection url. user-name,password etc.
Persistence Context
• **A persistence context is a set of managed entity object instances**.
• Persistence contexts are managed by an entity manager.
• The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database using the flush mode rules
• Once a persistence context is closed, all managed entity object instances become

detached and are no longer managed.
• Once an object is detached from a persistence context, it can no longer be managed by an entity manager, and any state changes to this object instance will not be synchronized with the database.
• When a persistence context is closed, all managed entity objects become detached and are unmanaged.
• There are two types of persistence contexts:
• transaction-scoped persistence context
• extended persistence context.

Transaction Scoped Persistence context

• Everything executed
• Either fully succeed or fully fail

6. Explain how to create bound properties of design patterns

A Bean containing a bound property must maintain a list of property change listeners, and alert those listeners when the bound property changes. The convenience class PropertyChangeSupport implements methods that add and remove PropertyChangeListener objects from a list, and fires PropertyChangeEvent objects at those listeners when the bound property changes. Your Beans can inherit from this class, or use it as an inner class.

An object that wants to listen for property changes must be able to add and remove itself from the listener list on the Bean containing the bound property, and respond to the event notification method that signals a property change. By implementing the PropertyChangeListener interface the listener can be added to the list maintained by the bound property Bean, and because it implements the PropertyChangeListener.propertyChange method, the listener can respond to property change notifications.

The PropertyChangeEvent class encapsulates property change information, and is sent from the property change event source to each object in the property change listener list via thepropertyChange method.

The following sections provide the details of implementing bound properties.

Implementing Bound Property Support Within a Bean

To implement a bound property, take the following steps:

Import the java.beans package. This gives you access to the PropertyChangeSupport class.

Instantiate a PropertyChangeSupport object:

```
private PropertyChangeSupport changes =
                    new PropertyChangeSupport(this);
```

This object maintains the property change listener list and fires property change events. You can also make your class a PropertyChangeSupport subclass.

Implement methods to maintain the property change listener list. Since PropertyChangeSupport implements these methods, you merely wrap calls to the property-change support object's methods:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
changes.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l) {
changes.removePropertyChangeListener(l);
}
```

Modify a property's setter method to fire a property change event when the property is changed. OurButton's setLabel method looks like this:

```
public void setLabel(String newLabel) {
String oldLabel = label;
label = newLabel;
sizeToFit();
changes.firePropertyChange("label", oldLabel, newLabel);
}
```

Note that setLabel stores the old label value, because both the old and new labels must be passed to firePropertyChange.

```
public void firePropertyChange(String propertyName,
Object oldValue, Object newValue)
```

The firePropertyChange method bundles its parameters into a PropertyChangeEvent object, and calls propertyChange(PropertyChangeEvent pce) on each registered listener. The old and new values are treated as Object values. If your property values are primitive types such as int, you must use the object wrapper version such as java.lang.Integer. Also, property change events are fired *after* the property has changed.

When the BeanBox (or Beans-aware builder tool) recognizes the design patterns for bound properties within your Bean, you will see a propertyChange interface item when you select the Edit|Events menu.

Now that you have given your Bean the ability to broadcast events when a bound property has changed, the next step is to create a listener.

Implementing Bound Property Listeners

To listen for property change events, your listener Bean must implement the PropertyChangeListener interface. This interface contains one method:

```
public abstract void propertyChange(PropertyChangeEvent evt)
```

This is the notification method that the source Bean calls on all property change listeners in its property change listener list.

So to make your class able to listen and respond to property change events, you must:

Implement the PropertyChangeListener interface.

```
public class MyClass implements java.beans.PropertyChangeListener,
                                java.io.Serializable {
```

Implement the propertyChange method in the listener. This method needs to contain the code that handles what you need to do when the listener receives property change event. Very often, for example, this is a call to a setter method in the listener class: a property change in the source Bean propagates a change to a property in a listener Bean.

To register interest in receiving notification about a Bean property change, the listener Bean calls the listener registration method on the source Bean. For example:

```
button.addPropertyChangeListener(aButtonListener);
```

Or, you can use an adapter class to catch the property change event, and subsequently call the correct method within the listener object. Here is an example taken from comments in the beans/demo/sunw/demo/misc/ChangeReporter.java file.

```
OurButton button = new OurButton();
...
PropertyChangeAdapter adapter = new PropertyChangeAdapter();
...
button.addPropertyChangeListener(adapter);
...
class PropertyChangeAdapter implements PropertyChangeListener
{
  public void propertyChange(PropertyChangeEvent e)
  {
    reporter.reportChange(e);
  }
}
```

Bound Properties in the BeanBox

The BeanBox handles bound property events as it handles all events: by using an event hookup adapter. Event hookup adapters classes are generated by builder tools when you connect an event source Bean to an event listener Bean. These objects interpose between event sources and event listeners to provide control and filtering over event delivery. Since an event listener can register with multiple listeners that fire the same event type, event hookup adapters can be used to intercept an event from a particular event source, and forward it to the correct event listener. This saves the event listener from implementing code that would examine each event to determine if it is from the correct source. See section 6.7 of the JavaBeans API Specification for a complete discussion of event hookup adapters.

The OurButton and ChangeReporter Beans can be used to illustrate this technique. To see how this works, take the following steps:

1. Drop OurButton and ChangeReporter instances on the BeanBox.
2. Select the OurButton instance and choose the Edit|Events|propertyChange|propertyChange menu item.
3. Connect the rubber band line to the ChangeReporter instance. The EventTargetDialog will be displayed.
4. Choose reportChange from the EventTargetDialog. The event hookup adapter source will be generated and compiled
5. Select OurButton and change some of it's properties. You will see change reports in ChangeReporter.

Behind the scenes the BeanBox generated the event hookup adapter. This adapter implements the PropertyChangeListener interface, and also generates a propertyChangemethod implementation that calls the ChangeReporter.reportChange method. Here's the generated adapter source code:

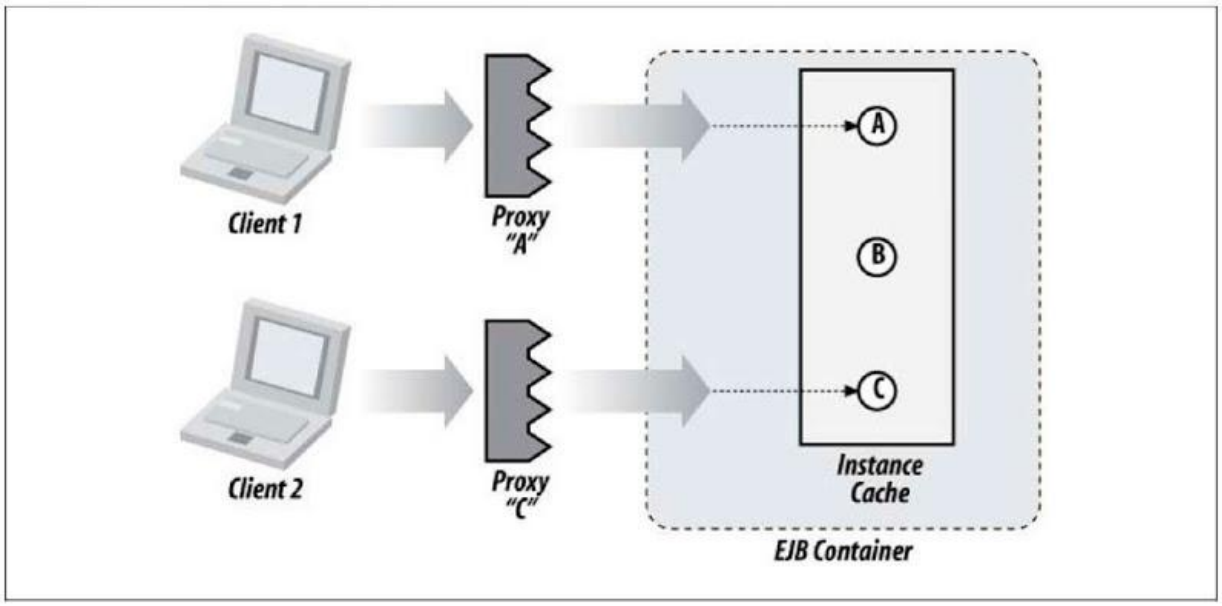
```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.misc.ChangeReporter;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

public class ___Hookup_14636f1560 implements
        java.beans.PropertyChangeListener, java.io.Serializable {

    public void setTarget(sunw.demo.misc.ChangeReporter t) {
        target = t;
    }

    public void propertyChange(java.beans.PropertyChangeEvent arg0) {
        target.reportChange(arg0);
    }

    private sunw.demo.misc.ChangeReporter target;
}
```

The ChangeReporter Bean need not implement the PropertyChangeListener interface; instead, the BeanBox generated adapter class implements PropertyChangeListener, and the adapter's propertyChange method calls the appropriate method in the target object (ChangeReporter).

The BeanBox puts the event adapter classes in the beans/beanbox/tmp/sunw/beanbox directory. When an adapter class is generated, you can view the adapter source in that directory.

7.(a) Explain about Stateful Session Bean

- While the strengths of the stateless session bean lie in its speed and efficiency, stateful session beans are built as a server-side extension of the client.
- Each SFSB is dedicated to one client for the life of the bean instance; it acts on behalf of that client as its agent (see Figure 6-1).
- Stateful session beans are not swapped among EJB objects, nor are they kept in an instance pool like their stateless session counterparts.
- Once a stateful session bean is instantiated and assigned to an EJB object, it is dedicated to that EJB object for its entire lifecycle.



Stateful session beans maintain conversational state, which means that the instance variables of the bean class can maintain data specific to the client between method invocations.
This makes it possible for methods to be interdependent such that changes made to the bean's state in one method call can affect the results of subsequent method invocations.
Therefore, every method call from a client must be serviced by the same instance (at least conceptually), so the bean instance's state can be predicted from one method invocation to the next.

In contrast, stateless session beans don't maintain client-specific data from one method call to the next, so any instance can be used to service any method call from any client.
* This is a conceptual model**.** Some EJB containers may actually use instance swapping with stateful session beans but make it appear as if the same instance is servicing all requests. Conceptually, however, the same stateful session bean instance services all requests.
Although stateful session beans maintain conversational state**,** they are not themselves persistent; the state of a SFSB is lost when the session is removed, the session times out, or the server restarts. Persistent state in EJB is modeled by the entity bean.
Because SFSBs are often considered extensions of the client**,** we may think of a client as being composed from a combination of operations and state. Each task may rely on some information gathered or changed by a previous operation. A GUI client is a perfect example: when you fill in the fields on a GUI client, you are creating conversational state. Pressing a button executes an operation that might fill in more fields, based on the information you entered previously. The information in the fields is conversational state.

Stateful session beans allow you to encapsulate some of the business logic and conversational state of a client and move it to the server. Moving this logic to the server thins the client application and makes the system as a whole easier to manage. The stateful session bean acts as an agent for the client, managing processes or taskflow to accomplish a set of tasks; it manages the interactions of other beans in addition to direct data access over several operations to accomplish a complex set of tasks. By encapsulating and managing taskflow on behalf of the client, stateful beans present a simplified interface that hides the details of many interdependent operations on the database and other beans from the client.

(b) Write a short note on Entity Relationship

Seven types of relationships can exist between entity beans. There are four types of cardinality: one-to-one, one-to-many, many-to-one, and many-to-many. In addition, each relationship can be either unidirectional or bidirectional. These options seem to yield eight possibilities, but if you think about it, you'll realize that one-to-many and many-to-one bidirectional relationships are actually the same thing. Thus, there are only seven distinct relationship types. To understand relationships, it helps to think about some simple examples:

One-to-one unidirectional

The relationship between an employee and an address. You clearly want to be able to look up an employee's address, but you probably don't care about looking up an address's employee.

One-to-one bidirectional

The relationship between an employee and a computer. Given an employee, we'll need to be able to look up the computer ID for tracing purposes. Assuming the computer is in the tech department for servicing, it's also helpful to locate the employee when all work is completed.

One-to-many unidirectional

The relationship between an employee and a phone number. An employee can have many phone numbers (business, home, cell, etc.). You might need to look up an employee's phone number, but you probably wouldn't use one of those numbers to look up the employee.

One-to-many bidirectional

The relationship between an employee (manager) and direct reports. Given a manager, we'd like to know who's working under him or her. Similarly, we'd like to be able to find the manager for a given employee.

Many-to-one unidirectional

The relationship between a customer and his or her primary employee contact. Given a customer, we'd like to know who's in charge of handling the account. It might be less useful to look up all the accounts a specific employee is fronting, although if you want this capability you can implement a many-to-one bidirectional relationship.

Many-to-many unidirectional

The relationship between employees and tasks to be completed. Each task may be assigned to a number of employees, and employees may be responsible for many tasks. For now we'll assume that given a task we need to find its related employees, but not the other way around.

Many-to-many bidirectional

The relationship between an employee and the teams to which he or she belongs. Teams may also have many employees, and we'd like to do lookups in both directions.