Web Component Development Using Java

Session: 5

Database Access and Event Handling

Objectives

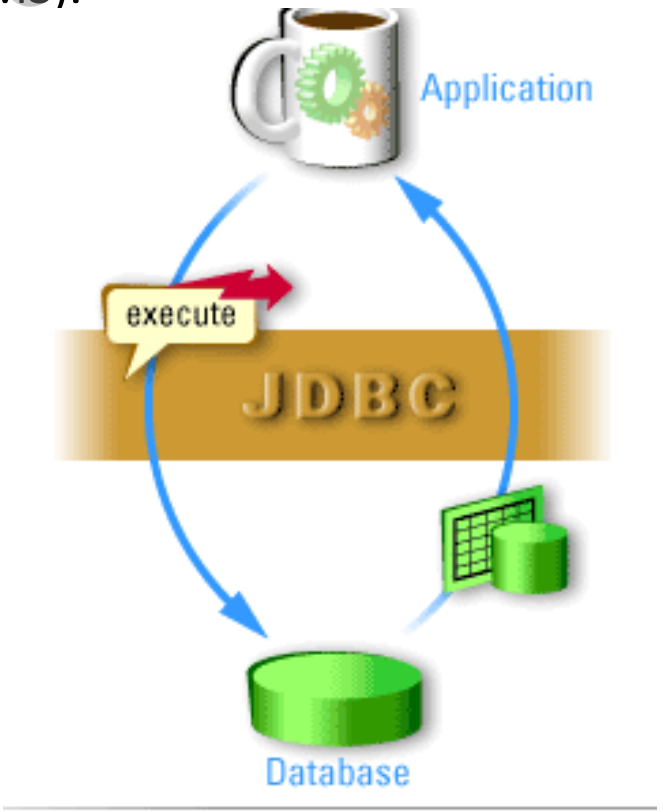
- Explain database handling using JDBC
- Describe JDBC and its role
- Explain connecting database using JDBC
- Describe JPA and its role
- Describe connecting database using JPA
- Explain the significance of session handling and session events
- Explain different types of listener interfaces used in Servlets

Introduction 1-2

Java provides various mechanisms using which data can be accessed from the database.

These are as follows:

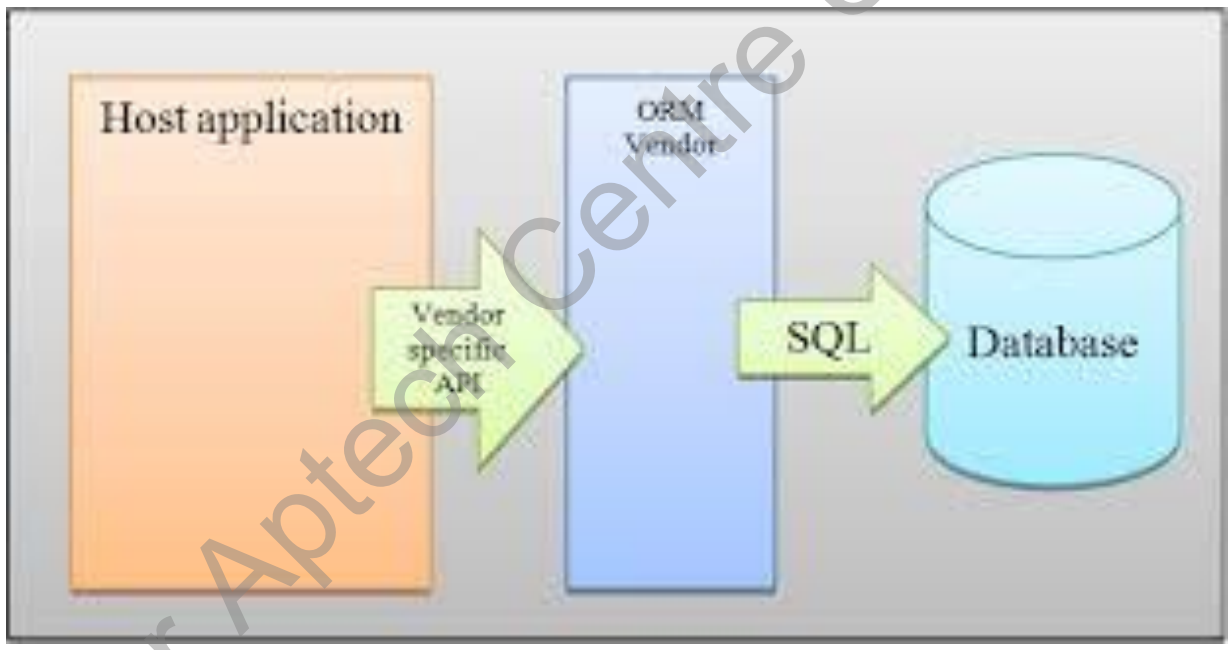
- **Using Java Database Connectivity (JDBC) API**
  - It is a part of Java SE platform that enables a Web application to interact with a Database Management System (DBMS).



Introduction 2-2

Using Object Relational Mapping (ORM) API

- It is a persistence mechanism used by enterprise application to access relational databases.
- It maps the Java objects to database tables through XML configuration.

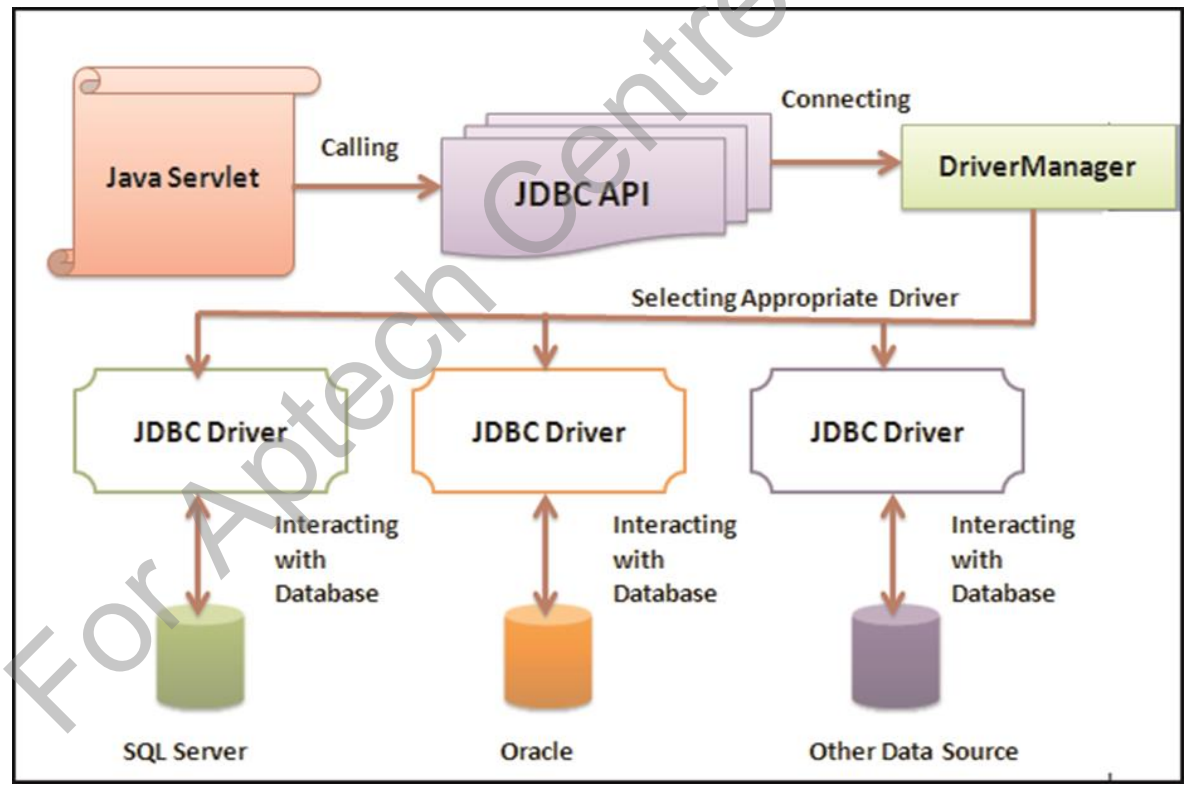


JDBC API

- Provides classes and interfaces that allow a Web application to access and perform operations on the databases.
- The operations that can be performed on the databases include:
  - Connecting to the databases
  - Processing SQL statements
  - Building result sets
  - Executing the parameterized statements
  - Invoking callable statements

JDBC Architecture

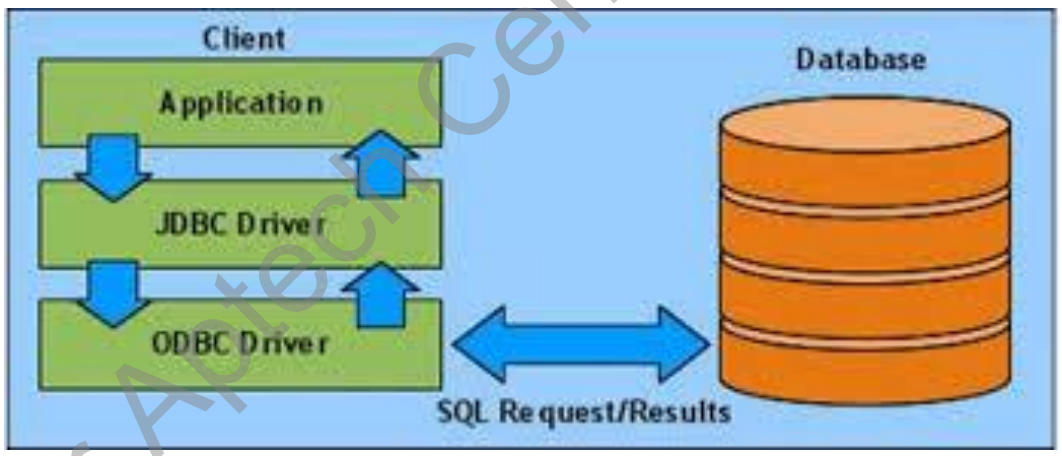
- Servlet uses JDBC APIs to access database.
- JDBC API uses a driver manager, which ensures that the correct driver is used for accessing a specific database.
- JDBC driver translates standard JDBC calls into a client's API specific calls.
- API calls facilitate communication with the database.



JDBC Driver Types 1-4

Type 1: JDBC-ODBC Bridge Plus ODBC Driver

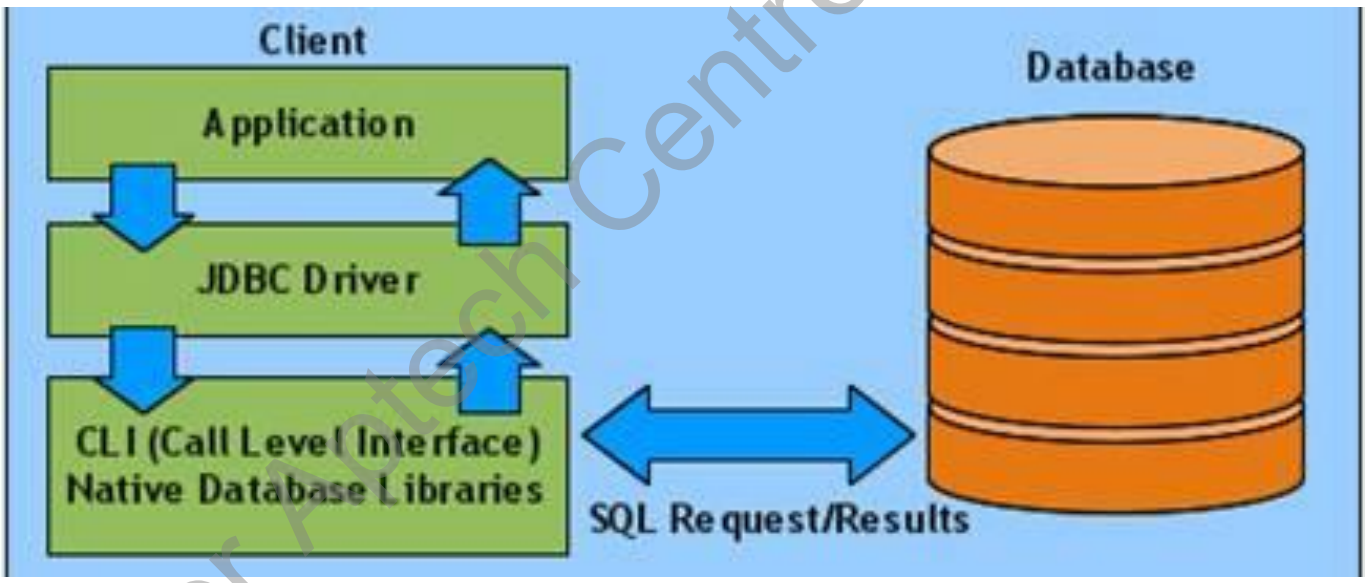
- Provides access to the database through standard ODBC libraries.
- Translates the standard JDBC calls to the corresponding ODBC calls.
- Part of the `sun.jdbc.odbc` package.
- Capabilities depend on the capabilities of the ODBC driver.
- Figure shows the JDBC-ODBC bridge driver.



JDBC Driver Types 2-4

Type 2: Native API Partly Java Driver

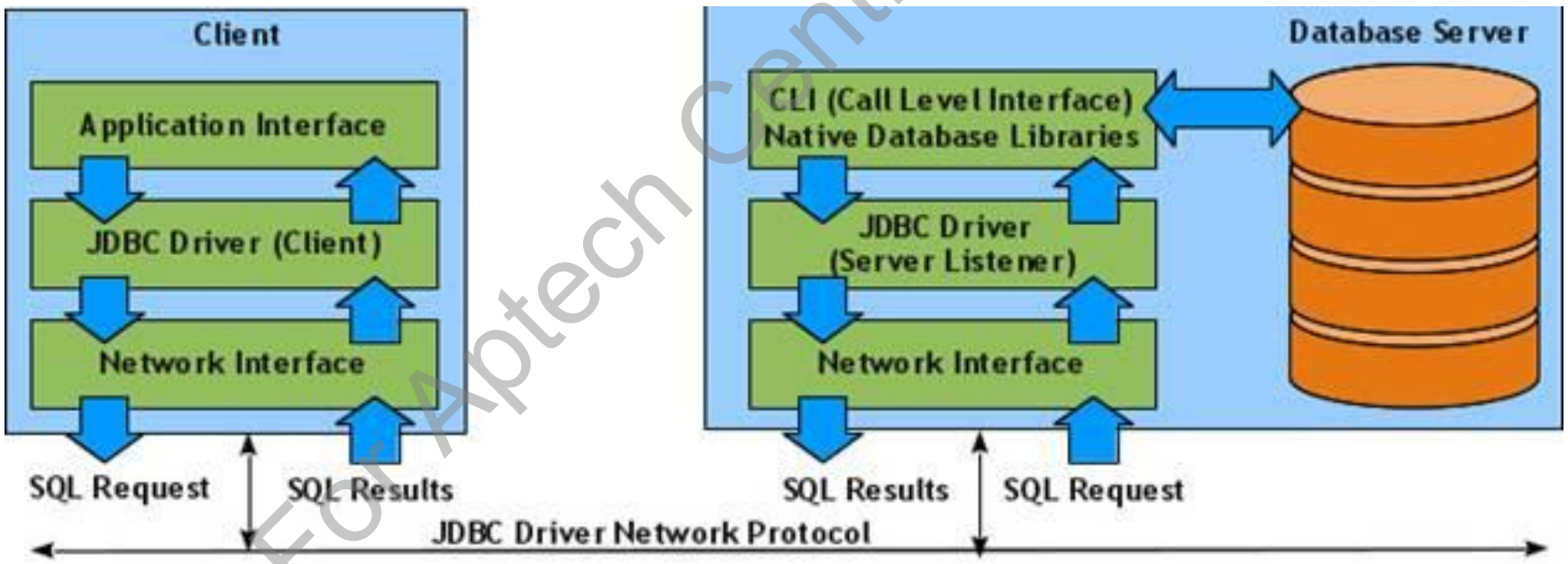
- Calls the native database library that accesses the database.
- Requires the native database libraries to be installed and configured on the client's machine.
- Figure shows native API partly Java driver.



JDBC Driver Types 3-4

Type 3: JDBC-Net Pure Java Driver

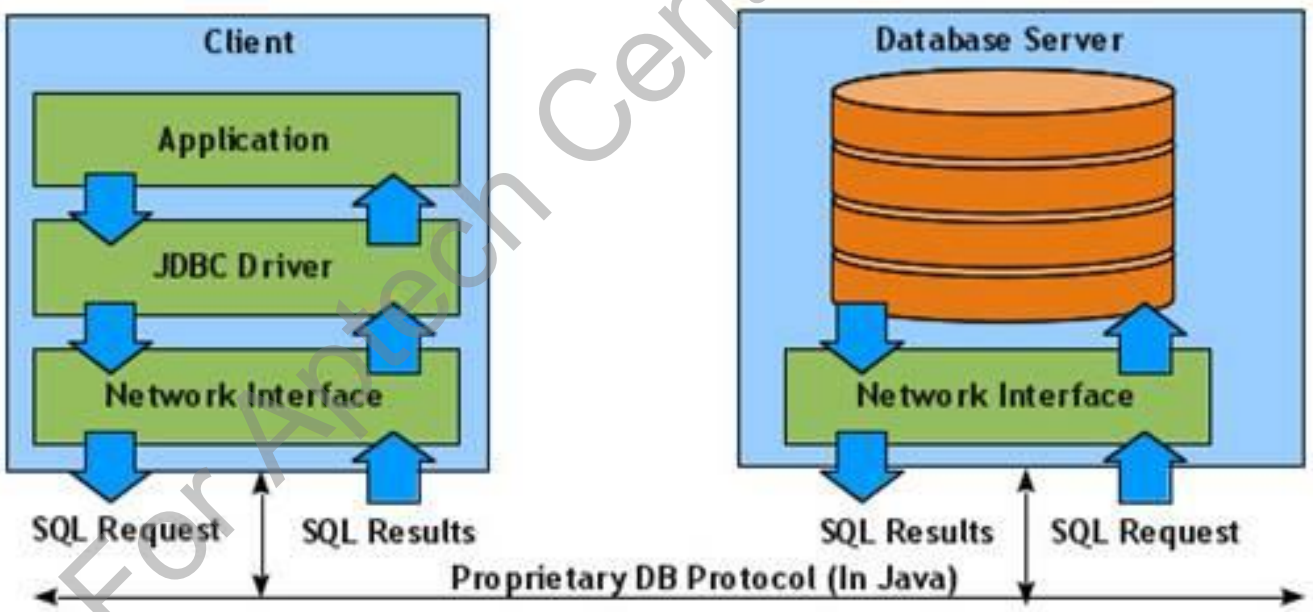
- Communicates with the JDBC middle tier on the server through a proprietary network protocol.
- Java Servlet sends a JDBC call to the middleware through the JDBC driver without translation, middleware then sends the request to the database.
- Figure shows JDBC-Net pure Java driver.



JDBC Driver Types 4-4

Type 4: Native-Protocol Pure Java Driver

- Fastest driver.
- Implements a proprietary database driver.
- Provides database connectivity through the standard JDBC APIs and do not require any native libraries on the client machine.
- Figure shows native-protocol pure Java driver.



Accessing Database Using JDBC

- Servlet interacts with the database using JDBC APIs and follows certain steps in sequence.

1. Registering the JDBC driver
2. Establishing a database connection
3. Executing an SQL statement
4. Processing the results
5. Closing the database connection

Registering the JDBC Driver

- Notifies the driver manager that a JDBC driver is available.
- JDBC automatically registers itself with the driver manager as soon as it is loaded.
- JDBC driver can be loaded at run time using the `forName()` method.
- **Syntax:**

```
Class.forName(String driver-name)
```

- The `forName()` is a static method that instructs the JVM to locate, load, and link the specified driver classes.

Establishing a Database Connection 1-2

- Identified by a database URL.
- Established using the `getConnection()` method of the `DriverManager` class in JDBC API.

- **Syntax for specifying the URL**:

```
jdbc:<subprotocol>:<database_identifier>
```

- where,
  - JDBC is used for establishing the connection.
  - `<subprotocol>` is the name of any valid database driver or any other database connectivity that is being requested.
  - `<database_identifier>` is the logical name of the database connection that maps to the physical database.

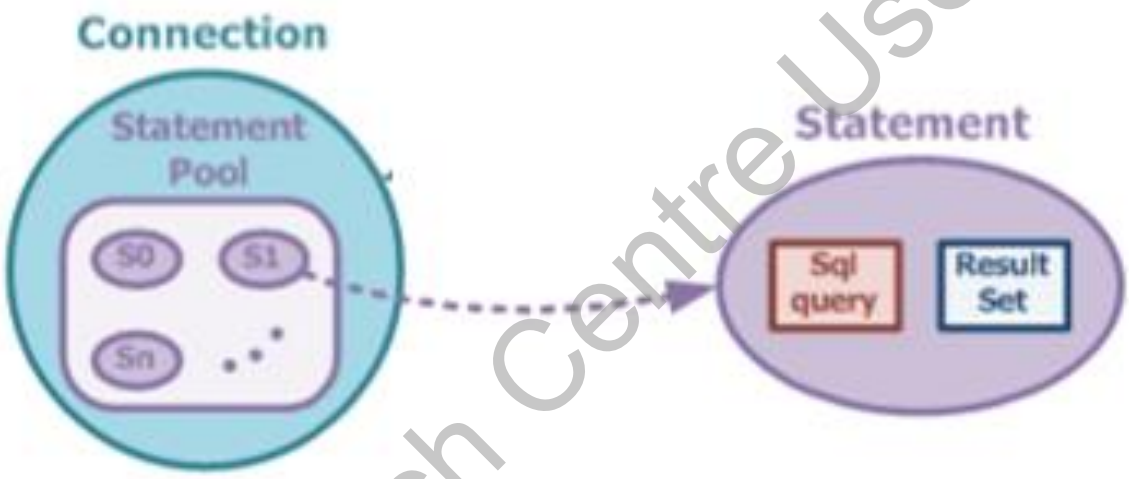
Establishing a Database Connection 2-2

- The code snippet demonstrates the connection to a SQL Server database from the Servlet.

```
public class JDBCServlet implements HttpServlet
{
   . . .
 Connection conn = null;
 try {
 // Loads Type 4 driver for connecting to SQL Server
//2012 database
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDri
ver");

// Obtains a connection to the database
conn =
DriverManager.getConnection("jdbc:sqlserver://localhost:
1433;"
+ "database=ProductStore;user=sa;password=sa;");
. . .
   } catch (ClassNotFoundException ex) {
          . . . }
. . .

}
```

Executing an SQL Statement 1-2

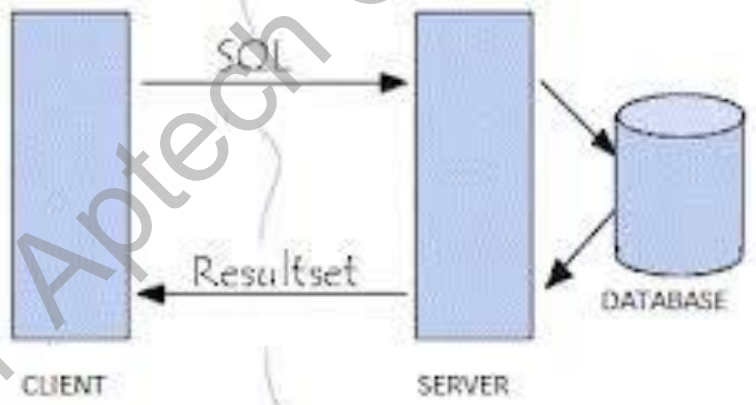
- To query the database, the `Statement` interface of JDBC API can be used.
- Figure shows the `Statement` object.



- The `Statement` object reference can be created using the `createStatement()` method on the obtained connection.
- For example, `Statement stat = conn.createStatement();` is used to create the `Statement` object.

Executing an SQL Statement 2-2

- The `executeQuery()` method of `Statement` interface accepts SQL query as argument and returns a `ResultSet` object.
- The `ResultSet` object contains all the rows returned by the SQL query.
- For example, `ResultSet rs = stat.executeQuery("SELECT * FROM Product");` shows the creation of query using `Statement` object.
- Figure shows the return of the `ResultSet` object on query execution.



Processing the ResultSet 1-2

- `ResultSet` object maintains a cursor which points to the current row in the result table.
- Three types of `ResultSet` are as follows:
  - `TYPE_FORWARD_ONLY` – cursor moves only in forward.
  - `TYPE_SCROLL_INSENSITIVE` – cursor scrolls forward and backward; not sensitive to changes, which have been made by other users to the database.
  - `TYPE_SCROLL_SENSITIVE` – cursor scrolls forward and backward; sensitive to changes, which have been made by other users to the database.

Processing the ResultSet 2-2

- The code snippet demonstrates how to iterate through the `resultset`.

```
// Access column values
while(rs.next()) {
      rs.getInt ("product_id");
      rs.getString ("product_name");
}
```

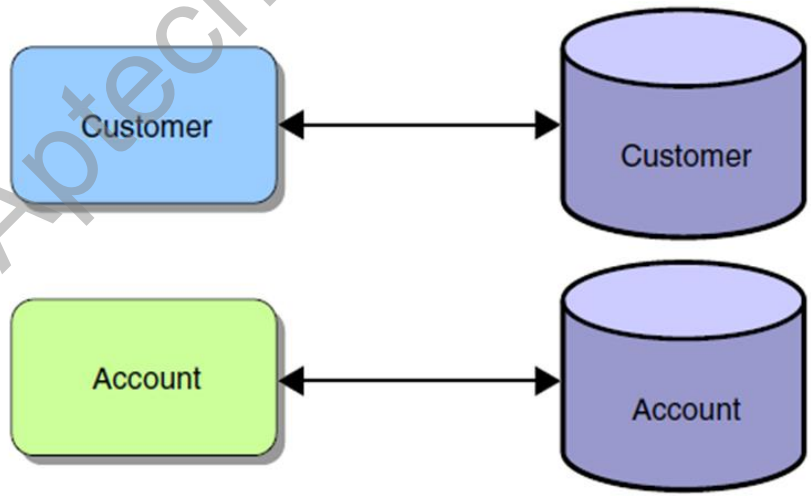
  - Iterating the `ResultSet` includes `next()` method which moves the cursor forward one row.
  - Returns true, if the cursor is now positioned on a row. Otherwise false, if the cursor is positioned after the last row.
- The code snippet demonstrates how to update the `resultset`.

```
rs.updateString("product_name", "Beverages");

rs.updateRow();
```

- The statement, `conn.close()` closes the connection.

Java Persistence API (JPA)

- **JPA:**
  - Lightweight, POJO-based Java framework.
  - Helps to persist the Java objects to the relational database.
  - Provides the persistence providers in the JPA application.
  - Converts data between incompatible type in object-oriented programming to the appropriate underlying database types.
- ORM technology that persist the entities in the database, supports the mapping to object-oriented software to the database schema through configuration mapping.
- Figure shows mapping of **Java Objects** to **Tables.**



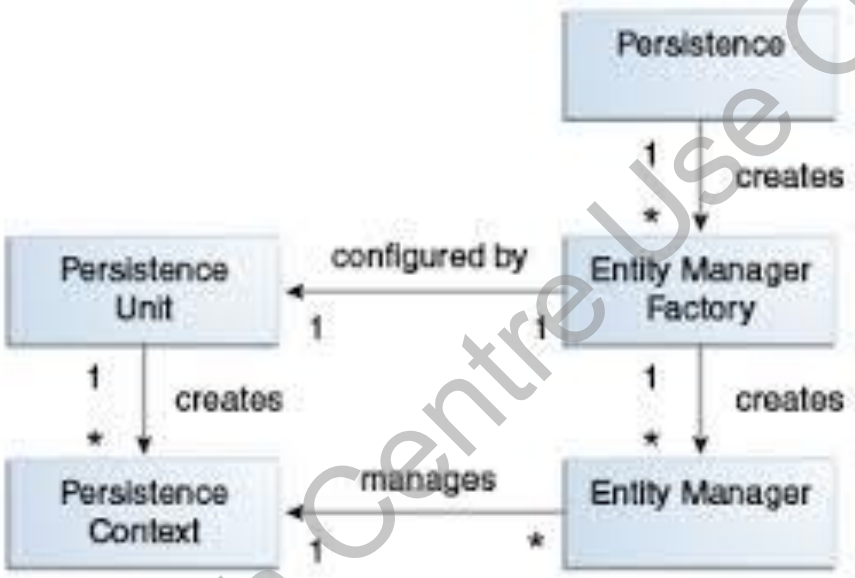
JPA Specification

- Features of JPA specification are as follows:
  - Supports POJO programming model for persisting objects into the database.
  - Provides a standard ORM API which incorporates concepts present in third-party persistence frameworks.
  - For example, Hibernate, Toplink, Java Data Object (JDO), and so on.
  - Defines a service provider interface implemented by different persistence providers.

POJOs persisted in the database by JPA are also referred as entities.

JPA Components

- Figure shows the primary components of JPA API.



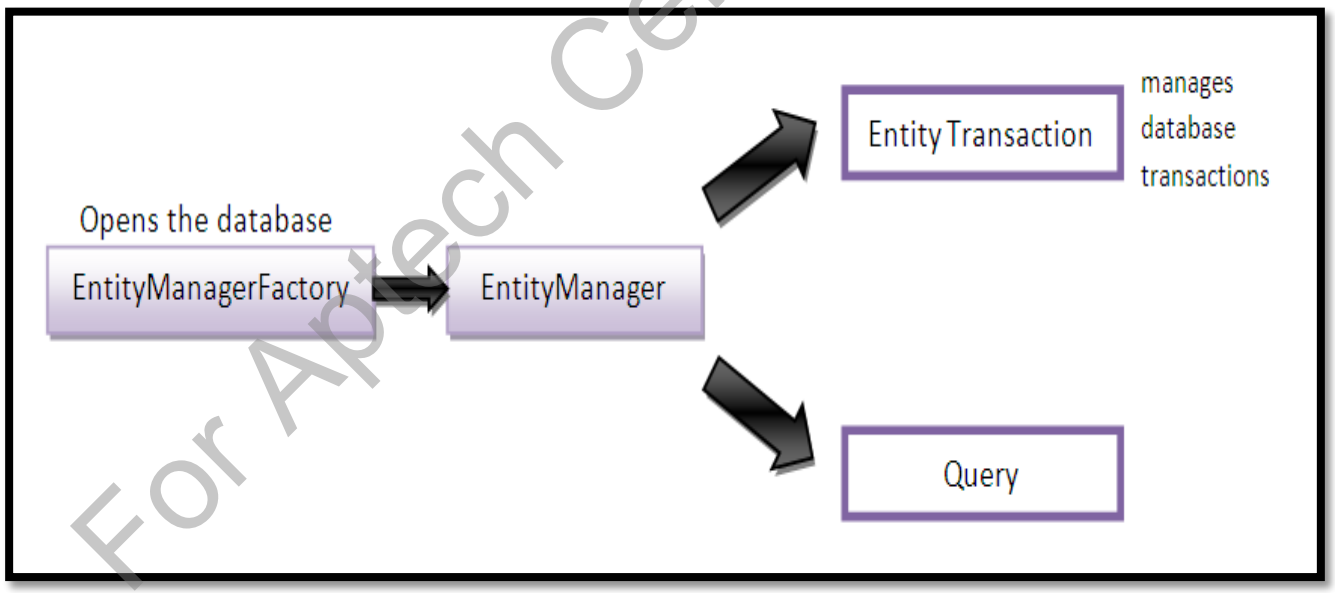
- **Persistence** - an identity which is a unique value used by the container to map the entity object to the corresponding record in the database.
- **EntityManager** - provides the API for interacting with the Entity.
- **EntityManagerFactory** - used to create an instance of `EntityManager`.
- **Persistence Unit** - consists of XML configuration information and a bundle of classes that are controlled by the JPA provider.
- **Persistence Context** - associated with entity manager, is a set of entities such that for any persistent identity, there is a unique entity instance.

Requirements of Entities Class in JPA

- The entity class must be annotated with `@Entity` annotation defined in `javax.persistence` package.
- The class must have a no-argument constructor with public or protected access modifier.
- The properties or methods of the entity class should not be declared as final. The class also should not be declared as final.
- If the entity instance is passed as parameter to a Session Bean remote interface, then it must implement the `Serializable` interface.
- Entities can be inherited from entity or non-entity classes.
- The instance variables of entity class must be declared as private or protected and are accessed only by the methods present in the class.
- Each entity instance must be identified by a unique identifier or a primary key. The primary key helps the client to access the entity instance.
- The `@Id` annotation applied on the entity property specifies that the property is a primary key.

Managing Entities in the Persistence Context 1-2

- To manage the entity instances in the persistence context, an interface named `EntityManager` API is used.
- The `EntityManager` interface is defined in `javax.persistence` package and is used to interact with the persistence context.
- The `EntityManager` interface provides methods to synchronize the state of the entity instance to the database.
- Figure shows the working of `EntityManager` in JPA.



Managing Entities in the Persistence Context 2-2

- Table lists some of the methods defined in the `EntityManager` interface.

Methods	Description
`public void persist (Object entity)`	The method saves an entity into the database and makes the entity managed.
`public <T> T merge (T entity)`	The method merges an entity to the EntityManager's persistent context and returns the merged entity.
`public void remove (Object entity)`	The method removes an entity from the database.
`public void flush()`	The method synchronizes the state of an entity in the EntityManager's persistent context with the database.

Packaging and Deploying Entity Classes 1-2

- **Persistence Unit:**
  - Is a set of class mapped to a particular database.
  - Is defined in a special descriptor file named `persistence.xml`.
  - Contains logical grouping of entity classes, their metadata mappings, and configurations related to a database.
  - The configuration details in the `persistence.xml` are used while obtaining the `EntityManager` instance in the client program.
  - The `EntityManager` instance accordingly persists the entity instances in the database.
- The code snippet displays the code for a `persistence.xml` file developed as part of the enterprise application.

```
...
<?xml version="1.0" encoding="UTF-8" ?>
<persistence
xmlns="http://java.sun.com/xml/ns/persistence">
<persistence-unit name="intro" />
</persistence>
...
```

Packaging and Deploying Entity Classes 2-2

- Some of the attributes defined in the `<persistence-unit>` element are as follows:
  - **`<description>`** - describes the persistence unit and is optional.
  - **`<provider>`** - must be present in the J2SE environment or when the application requires a provider specific behavior.
  - **`<transaction-type>`** - has a value either as Java Transaction API (JTA) or RESOURCE_LOCAL or by default, the value is JTA.
  - **`<jta-data-source>/<non-jta-data-source>`** - used for specifying the Java Naming and Directory Interface (JNDI) name of the data source. JNDI is used by the persistence provider.
  - **`<mapping-file>`** - contains a list of one or more additional XML files that are used for O/R mapping. The mapping file is used to list the entity classes that are available in the persistence unit.
  - **`<properties>`** - specifies the configuration properties that are vendor-specific for the persistence unit. Any properties that are not recognized by the persistence provider are ignored.

Accessing Database Using JPA 1-8

- The code snippet shows a JSP file, `addPlayers.jsp`, with a form to add new records to the list.

```
. . .
 <body>
  <h1>New Player record</h1>
  <form id="addNewPlayersForm" action="addPlayers"
method="post">

  <table>
    <tr><td>Rank</td><td><input type="text" id = "rank"
name="rank" /></td></tr>

    <tr><td>Player</td><td><input type="text" id =
"playerName " name="playerName " /></td></tr>

    <tr><td>Team</td><td><input type="text" id = "team"
name="team" /></td></tr>
  </table>

  <input type="submit" id="CreateRecord"
value="CreateRecord" />
  </form>
<a href="DisplayPlayers"><b>Complete List</b></a>
</body>
</html>
```

Accessing Database Using JPA 2-8

- The code snippet shows the code to display the records, `DisplayPlayers.jsp`, which are also accessible from other pages.

```
  <body>

    <h1>ICC <b>ODI</b> Players Ranking List</h1>

    <table id=" PlayerTBList" border="2" bgcolor="#B0B0B0">
<tr >
  <th>Rank</th>
  <th>Player</th>
  <th>Team</th>
</tr>
<c:forEach var="cricketer" begin="0"
items="${requestScope.playerList}">
<tr>
  <td>${cricketer.rank}  </td>
  <td>${cricketer.playerName }  </td>
  <td>${cricketer.team}  </td>
</tr>

</c:forEach>

</table>
<a href="addPlayers.jsp"><b>Add New Player Record</b></a>
</body>
</html>
```

Accessing Database Using JPA 3-8

- The code snippet shows the `index.jsp` page.

```
<html>
  <head>
  </head>
  <body>
  <jsp:forward page="DisplayPlayers" />
  </body>
</html>
```

Accessing Database Using JPA 4-8

- The code snippet shows the entity class named `Players`.

```
@Entity
@Table(name = "PLAYERS")
public class Players {
   // column mapping
  @Rank
  @Column(name = "RANK")
  private String rank;
  @Column(name = "PLAYER")
  private String playerName;
  @Column(name = "TEAM")
  private String team;

  public Players() {  }
  public Players(String rank, String playerName, String team)
   {
    this.rank = rank; his.playerName = playerName;
    this.team = team;
   }

  public String getRank() {
    return this.id;
  }
  public String getPlayer() {
    return this.playerName;
  }
  public String getTeam () {
    return this.team;
  }
}
```

Accessing Database Using JPA 5-8

- The code snippet shows the Servlet, `addNewPlayersServlet.java`, for adding new players.

```
. . .

protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException {

    assert eManagerFactory != null;
    EntityManager eManager = null;
    try {

      // the data from user's form
      String rank = (String) request.getParameter("rank");
      String playerName = (String)
request.getParameter("playerName ");
      String team = (String) request.getParameter("team");
```

Accessing Database Using JPA 6-8

```
        // player instance
        Players player = new Players(rank, playerName, team);

        // begin transaction uTransaction
        uTransation.begin();

        // since the eManager is created inside a transaction, it is
associated with the transaction
        eManager = eManagerFactory.createEntityManager();

        //persist the player entity
        eManager.persist(player);

        // commit transaction uTransaction
        uTransation.commit();

request.getRequestDispatcher("DisplayPlayers").forward(request,
response);
    } catch (Exception ex) {

          . . .
   }
    }
  }
```

Accessing Database Using JPA 7-8

- The code snippet shows the code for displaying players data in the Servlet, `DisplayPlayersServlet.java`.

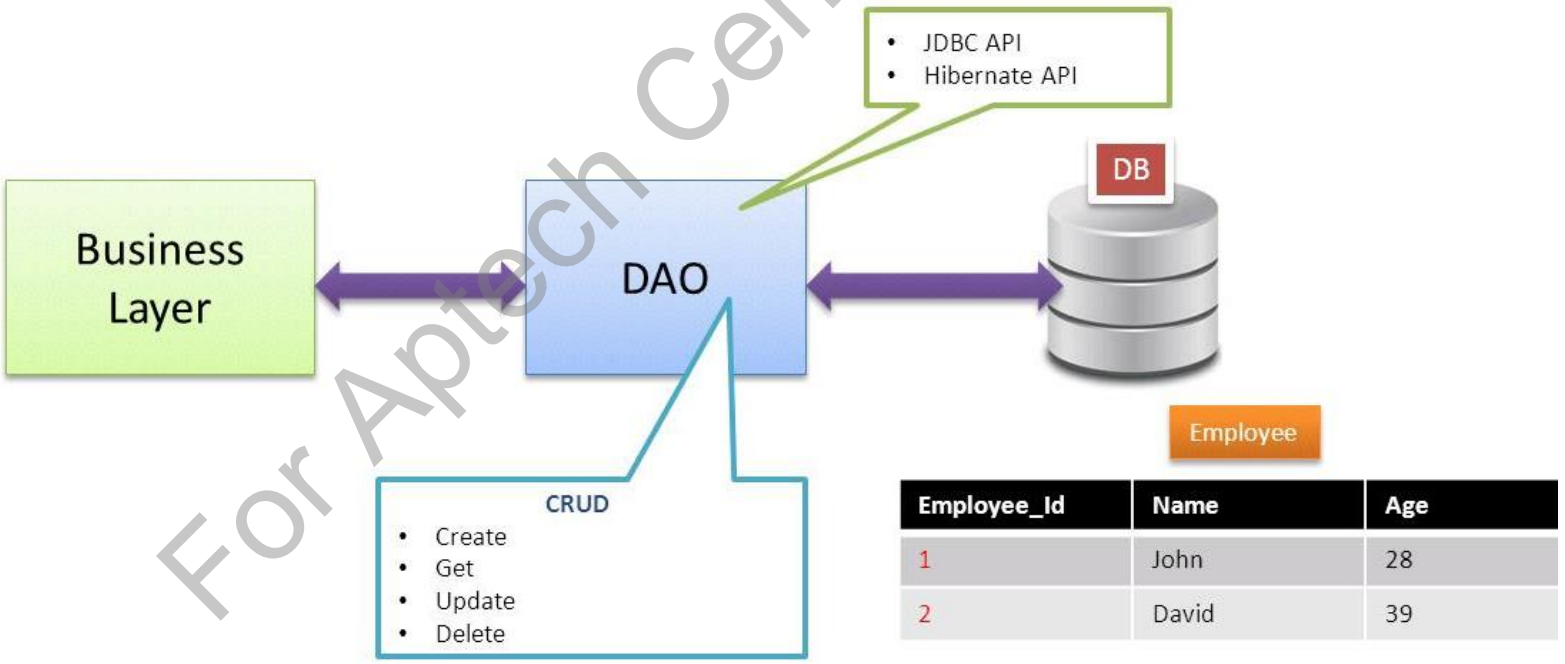
```
. . .
    EntityManager eManager = null;
    try {
      eManager = eManagerFactory.createEntityManager();

   //query for all the players in database
      List players = eManager.createQuery("select p from Player
p").getResultList();
      request.setAttribute("playerList",players);

      //Forward to the jsp page for rendering

request.getRequestDispatcher("DisplayPlayers.jsp").forward(requ
est, response);

    } catch (Exception ex) {
      throw new ServletException(ex);
    } finally {
      //close the eManager to release any resources held up by
the persistence provider
      if(eManager != null) {
        eManager.close();
      }
    }

  }
```

Accessing Database Using JPA 8-8

- Figures display the JSP page with the index page and other pages in the execution of the players application.

DAO Pattern for Databases 1-2

- Provides the easy maintenance of the applications by separating the business logics from the database access.
- Implementation of data access based on JDBC API is encapsulated in the DAO classes.
- The business tier in the Web application includes business service classes, domain object classes, and the DAO classes. The business service uses the DAO to perform the data access functions such as inserting or retrieving an object from the database.
- Figure shows the implementation of DAO pattern.



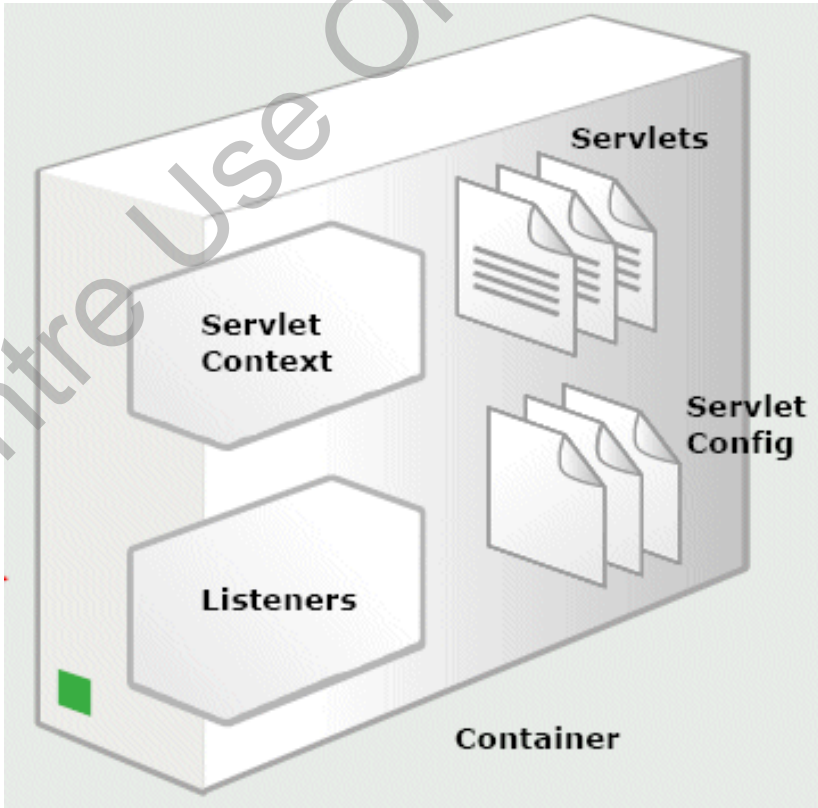
Employee_Id	Name	Age
1	John	28
2	David	39

DAO Pattern for Databases 2-2

- The advantages of DAO pattern are as follows:
  - Separation of domain objects and persistence logic.
  - Achieving flexibility and reusability in changing the data access.
  - Different types of clients such as Servlets or normal Java classes can use the data access logic provided in the DAO classes.
  - Provides independence to change the front-end technologies as well as backend database systems.

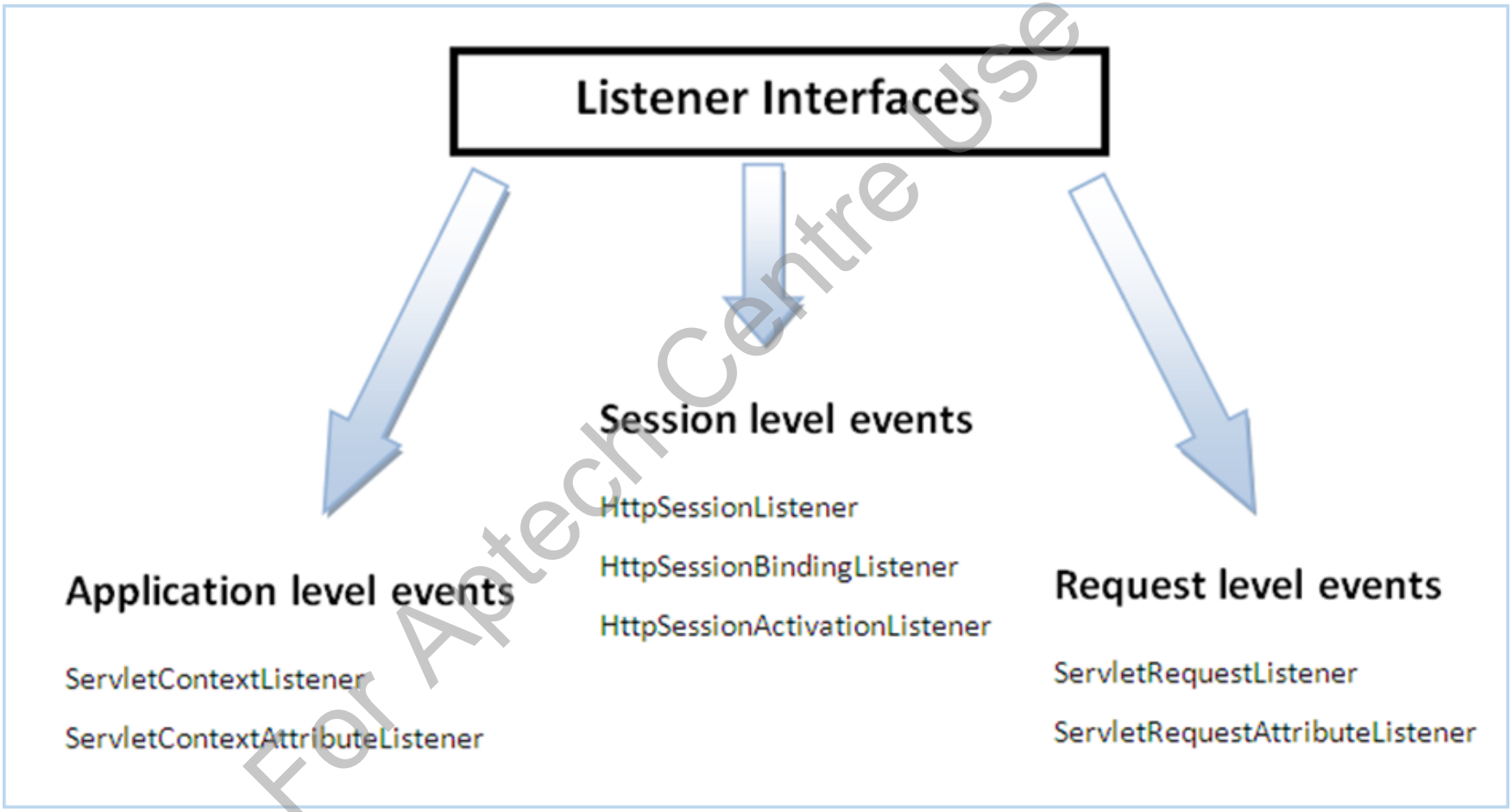
Session Event Handling

- Session handling is used to maintain the state of a user till the lifetime of the session.
- In Java EE,
  - The Web container transfers control to event handlers i.e. event listeners to listen events related to Servlets, JSP, and so on.



Listener Types

- The `javax.servlet` package defines the listener interfaces.
- Figure shows the three types of interfaces.



Handling Servlet Lifecycle Events

- The events in a servlet life cycle is monitored by defining listener objects.
- The listener objects methods get invoked when life cycle events occur.
- Follow these steps to use listener objects.
  - Define the listener class.
  - Mention the listener class in `web.xml`, that is, the Web application deployment descriptor.

ServletContextListener

- The methods present in the `ServletContextListener` class are as follows:
  - **`contextInitialized()`** - returns the notification that a Web application initialization has started.
  - **`contextDestroyed()`** - returns the notification about closing of the `servletcontext`. All servlets and filters are closed before notification.

- The code snippet shows the use of `ServletContext` methods.

```
public void contextInitialized(ServletContextEvent
event){

  ServletContext context = event.getServletContext();

  String IP = "10.1.1.142";
  context.setAttribute("DefaultAddress", IP);
}

public void contextDestroyed(ServletContextEvent event){
  ServletContext context = event.getServletContext();
  String IP = context.getAttribute("DefaultAddress");
}
```

ServletContextEvent 1-2

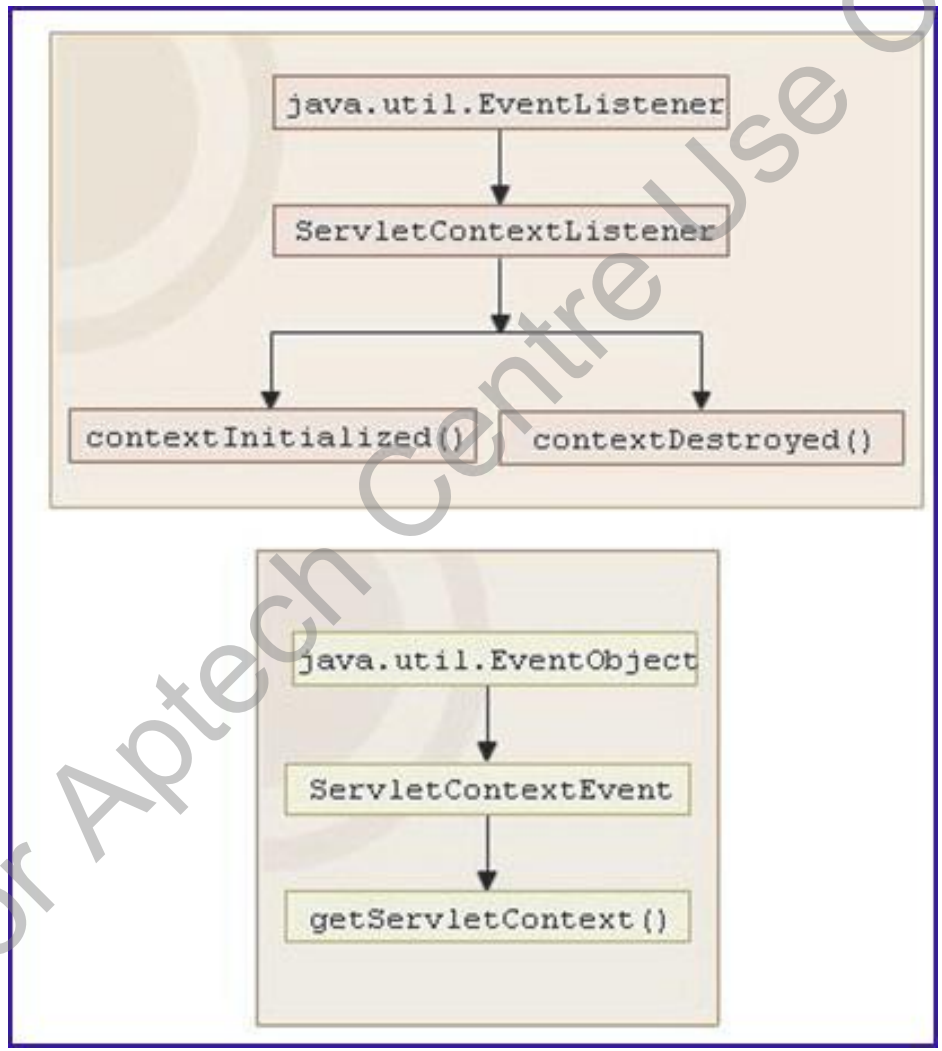
- The method present in `ServletContextEvent` class is as follows:
  - **`getServletContext()`** - returns the modified servlet context.

- The code snippet shows the use of the method `getServletContext()`.

```
// Save and get an application-scoped value
getServletContext().setAttribute("app-param", "app-
value1");


value1 = getServletContext().getAttribute("app-param");
```

ServletContextEvent 2-2

- Figures depict the hierarchy for `ServletContextListener`.



ServletContextAttributeListener 1-3

- The methods present in the interface are as follows:
  - **`attributeAdded()`**
    - It notifies that a new attribute is added to the `ServletContext`.
    - It is invoked after an attribute has been added.
  - **`attributeRemoved()`**
    - It notifies that an attribute is removed from `ServletContext`.
    - It is invoked after an attribute has been removed.

ServletContextAttributeListener 2-3

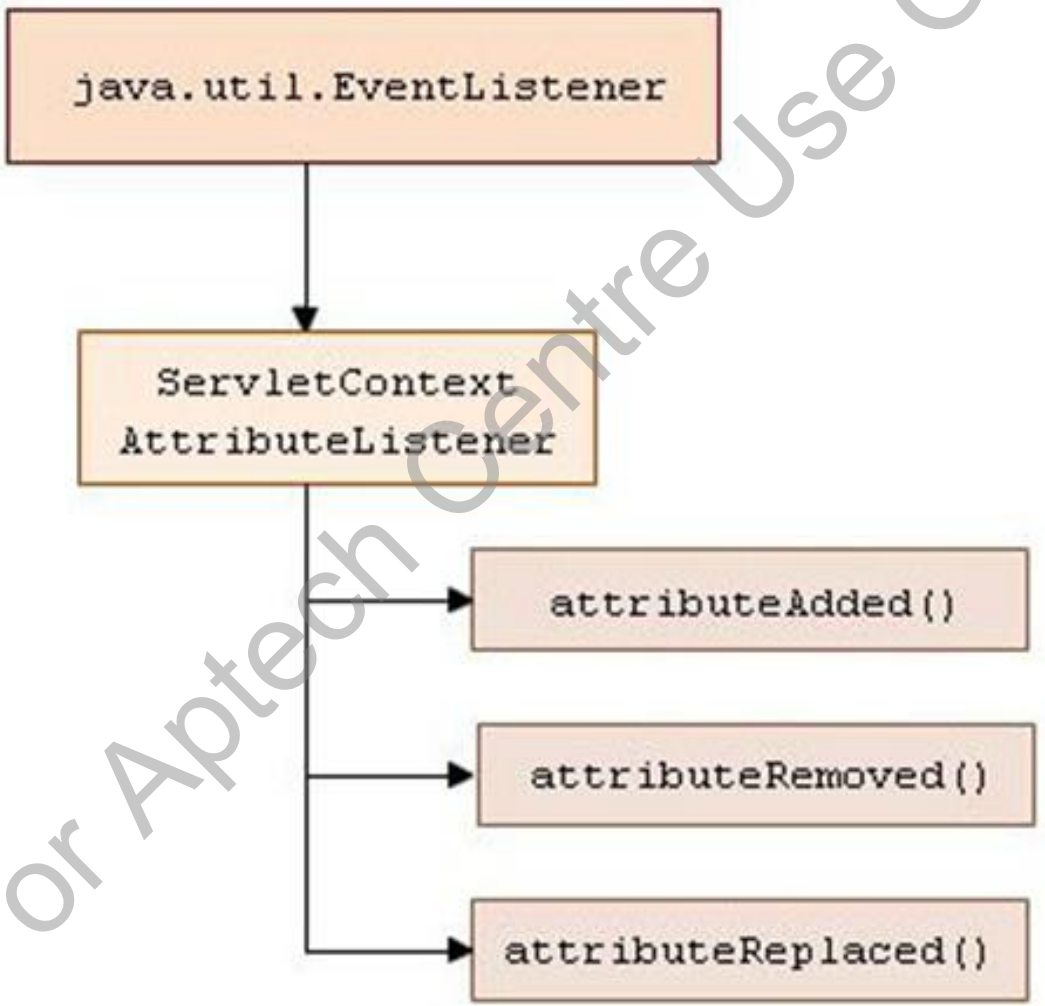
- The code snippet shows the method of the `ServletContextAttributeListener`.

```
public void attributeAdded(ServletContextAttributeEvent
event1){

  log.append(event1.getName() + "," + event.getValue()
+   "," + new Date() + "\n");

}

public void attributeRemoved
(ServletContextAttributeEvent event){
    log.append(event.getName() + "," + event.getValue()
+value1 =     "," + new Date() + "\n");
}
```

- **`attributeReplaced()`** - notifies that an attribute of `ServletContext` has been replaced. The method is invoked after an attribute has been replaced.

ServletContextAttributeListener 3-3

- Figure depicts the methods of `ServletContextAttributeListener`.



ServletContextAttributeEvent 1-2

- The methods present in the class are as follows:
  - **`getName()`** - name of the altered attribute in the `ServletContext` is returned.
  - The code snippet gets the name of the attribute and checks whether the attribute is altered.

```
public void attributeRemoved(ServletContextAttributeEvent
scae1){

  if (scae1.getName().equals("heading")) {

    ServletContext context1 =
                    scae1.getServletContext();

    String sHeading1=(String)
                context1.getAttribute("heading");

    context.log("Heading Replaced="+sHeading1);
  }
}
```

ServletContextAttributeEvent 2-2

- **`getValue()`** - attribute value that has been added, removed, or replaced is returned.
- The code snippet gets the value of the attribute and checks whether the value is not equal to null.

```
HttpSession session = request.getSession(true);
ShoppingCart1 previousItems1 =
(ShoppingCart1)session.getValue("previousItems1")
;

if (previousItems1 != null) {

   doSomethingWith(previousItems1);

} else {

   previousItems1 = new ShoppingCart1(...);
   doSomethingElseWith(previousItems1);
}
```

HttpSessionAttributeListener 1-2

The methods belonging to this interface are as follows:

- **`attributeAdded()`** – notifies whenever there is an addition of new attributes to a session, and is called after the addition of attribute.

- **`attributeRemoved()`** – notifies whenever there is a removal of attributes from a session, and is called after the removal of attribute.

HttpSessionAttributeListener 2-2

- **`attributeReplaced()`** - notifies whenever attributes are replaced in a session, and is called after the attribute is replaced.

- The code snippet implements `HttpSessionAttributeListener` interface and its methods.

```
public class UserAttributeListener implements
HttpSessionAttributeListener{

// declaring attributeAdded() method
public void attributeAdded(HttpSessionBindingEvent Demoevent)
{

// Logging the event details
  HTTPSession session = Demoevent.getSession();
  ServletContext sc = session.getServletContext();
  sc.log(Demoevent.getName() + "," +
                        Demoevent.getValue() );
 }

// Declaring attributeRemoved() method
public void attributeRemoved(HttpSessionBindingEvent Demoevent){}

// Declaring attributeReplaced() method
 public void attributeReplaced(HttpSessionBindingEvent Demoevent){}
}
```

HttpSessionBindingListener

The methods belonging to this interface are as follows:

- **valueBound()** - notifies the object on being bound to a session and is responsible for identification of the session.
- **valueUnbound()** - notifies the object on being unbound from a session and is responsible for identification of the session.

- The code snippet implements the `valueBound` and `valueUnbound` methods of `HttpSessionBindingListener` interface.

```
public class UserInfo implements HttpSessionBindingListener{
  private String uName;
  public User(String userName){
    uName = userName;
  }
  public void valueBound(HttpSessionBindingEvent ev){
    System.out.println(uName+" user bound");
  }
  public void valueUnbound(HttpSessionBindingEvent ev){
    System.out.println(uName+" user unbound");
  }
}
```

HttpSessionBindingEvent 1-3

The methods belonging to this class are as follows:

- **`getName()`** - returns a string specifying the name with which the attribute is bound to or unbound from the session.

- The code snippet gets the name string specifying the name with which the attribute is bound to or unbound from the session.

```
private void checkAttribute(HttpSessionBindingEvent event,
String orderAttributeName, String keyItemName,
String message) {


String demoCurrentAttributeName = event.getName();
String demoCurrentItemName = (String)event.getValue();

if (demoCurrentAttributeName.equals(orderAttributeName) &&
demoCurrentItemName.equals(keyItemName)) {

   ServletContext context =
   event.getSession().getServletContext();
   context.log("Customer" + message + keyItemName + ".");
}
```

HttpSessionBindingEvent 2-3

- **`getSession()`** - returns the session object that has changed.
- The code snippet returns the session object that has changed.

```
public void sessionCreated(HttpSessionBindingEvent event)
{

 // Retrieves modified session object
 HttpSession session = event.getSession();

     session.getServletContext().log("SessionListener01:
sessionCreated(" + session.getId() + ")");

   }
```

HttpSessionBindingEvent 3-3

- **`getValue()`** - returns the value of the attribute that has been added, removed, or replaced.

- The code snippet returns the value of the attribute that has been added, removed, or replaced.

```
private void checkAttribute(HttpSessionBindingEvent event,
String orderAttributeName, String keyItemName,
String message) {

String demoCurrentAttributeName = event.getName();
String demoCurrentItemName = (String)event.getValue();


if (demoCurrentAttributeName.equals(orderAttributeName) &&
demoCurrentItemName.equals(keyItemName)) {

   ServletContext context =
   event.getSession().getServletContext();
   context.log("Customer" + message + keyItemName + ".");
}
```

HttpSessionListener 1-2

The methods belonging to this interface are as follows:

- **`sessionCreated(HttpSessionEvent se)`** – provides notification that a session was created.

- The code snippet implements `HttpSessionListener` interface and its methods.

```
public class SessionTracker extends HttpServlet
implements HttpSessionListener{

static private int sessionCount; // count of session

// Implementing sessionCreated()

public void sessionCreated(HttpSessionEvent event){
    sessionCount++;
  }
```

HttpSessionListener 2-2

- **`sessionDestroyed(HttpSessionEvent se)`** – provides notification that a session is about to be invalidated.

- The code snippet implements the method `SessionDestroyed()` of the `HttpSessionListener` interface.

```
public void sessionDestroyed(HttpSessionEvent se) {
      HttpSession session = se.getSession();
   try{       session.invalidate();
    }
    catch(IllegalStateException e) { }
   }
}
```

- The `SessionTracker` class must be configured in the deployment descriptor of the Web application as shown in the code snippet.

```
<listener>
  <listener-class>
  SessionTracker
  </listener-class>
</listener>
```

HttpSessionActivationListener

The methods present in the listener are as follows:

- **`sessionDidActivate(HttpSessionEvent se)`** – provides notification that the session has just been activated.

- **`sessionWillPassivate(HttpSessionEvent se)`** – provides notification that the session is about to be passivated.

- The code snippet implements `HttpSessionActivationListener` interface and its methods.

```
public class SessionTracker implements
HttpSessionActivationListener {
public void sessionWillPassivate(HttpSessionEvent se) {
        System.out.println("session is about to be passivated");
        }

public void sessionDidActivate(HttpSessionEvent se) {
        System.out.println("session has just been activated");

        }
}
```

HttpSessionEvent

The method belonging to this class is `getSession()`.

- **`getSession`** – returns the session that changed within a Web application.

- The code snippet shows the `getSession()` method.

```
// returns modified session object//
public HttpSession getSession()
{
  return session;
}
```

Use of Listeners for Handling Events

- Event listeners should be executed quickly, because all event listening and drawing methods are executed in the same thread.
- While designing the program, one should implement their Event Listeners in a class that is not public.
- The code snippet shows the implementation of `actionPerformed()` method.

```
public class Beeper ... implements ActionListener {
  ...
//here initialization:
    button.addActionListener(this);
  ...
  public void actionPerformed(ActionEvent e) {
    ...
//Make a beep sound...
  }
}
```

Summary

- The most common operation performed by the Servlet is storing and retrieving database information.
- Java provides various mechanisms using which data can be accessed from the database using JDBC or JPA API.
- The JDBC API provides classes and interfaces that allow a Web application to access and perform operations on the databases.
- JPA is an ORM technology that persist the entities in the database.
- Java annotations or XML are used to define the mapping of entity to existing database tables.
- In JPA, persistent objects are referred as Entities. Entities are plain old Java objects that are persisted to relational databases or legacy systems.
- A persistence context represents a set of managed entity instances that exist in a particular data store.
- The DAO pattern provides the easy maintenance of the applications by separating the business logics from the database access.
- The events in a servlet life cycle is monitored by defining listener objects. These objects methods get invoked when life cycle events occur.