

What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs.

Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listeners process the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listeners that want to receive them.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represent an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
 - `public void addActionListener(ActionListener a){ }`
- **MenuItem**
 - `public void addActionListener(ActionListener a){ }`
- **TextField**
 - `public void addActionListener(ActionListener a){ }`
 - `public void addTextListener(TextListener a){ }`
- **TextArea**
 - `public void addTextListener(TextListener a){ }`
- **Checkbox**
 - `public void addItemListener(ItemListener a){ }`
- **Choice**
 - `public void addItemListener(ItemListener a){ }`
- **List**
 - `public void addActionListener(ActionListener a){ }`
 - `public void addItemListener(ItemListener a){ }`

What is JDBC?

JDBC is an acronym for Java Database Connectivity. JDBC is an standard API specification developed in order to move data from frontend to backend. This API consists of classes and interfaces written in Java. It basically acts as an interface or channel between our Java program and databases i.e it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use.

Establishing JDBC Connection in Java

Steps for connectivity between Java program and database

1. Loading the Driver

To begin with, we first need load the driver or register it before using it in the program . Registration is to be done once in our program. We can register a driver in one of two ways mentioned below :

- a. **Class.forName()** : Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- b. **DriverManager.registerDriver()**: DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time. The following example uses DriverManager.registerDriver()to register the Oracle driver –

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

2. Create the connections

After loading the driver, establish connections using :

```
Connection con = DriverManager.getConnection(url,user,password)
```

- **user** – username from which our sql command prompt can be accessed.
- **password** – password from which our sql command prompt can be accessed.
- **con**: is a reference to Connection interface.
- **url** : Uniform Resource Locator. It can be created as follows:
- String url = " jdbc:oracle:thin:@localhost:1521:xe"

where oracle is the database used, thin is the driver used, @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function.

3. Create a statement:

Once a connection is established, we can interact with the database. The JDBC Statement, Callable Statement, and Prepared Statement interfaces define the methods that enable us to send SQL commands and receive data from our database. Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step.

4. **Execute the query**

Now comes the most important part i.e executing the query. Query here is an SQL Query. Some types of queries are as follows:

- Query for updating / inserting table in a database.
- Query for retrieving data .

The `executeQuery()` method of `Statement` interface is used to execute queries of retrieving values from the database. This method returns the object of `ResultSet` that can be used to get all the records of a table.

The `executeUpdate(sql query)` method of `Statement` interface is used to execute queries of updating/inserting .

Example:

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully : "+sql);
else
    System.out.println("insertion failed");
```

5. **Close the connections**

By closing connection, objects of `Statement` and `ResultSet` will be closed automatically.

The `close()` method of `Connection` interface is used to close the connection.

Example :

```
con.close();
```

Implementation

```
import java.sql.*;
import java.util.*;
class Main
{
    public static void main(String a[])
    {
        //Creating the connection
        String url = "jdbc:oracle:thin:@localhost:1521:xe";
        String user = "system";
        String pass = "12345";

        //Entering the data
        Scanner k = new Scanner(System.in);
        System.out.println("enter name");
        String name = k.next();
        System.out.println("enter roll no");
        int roll = k.nextInt();
        System.out.println("enter class");
```

```

String cls = k.next();

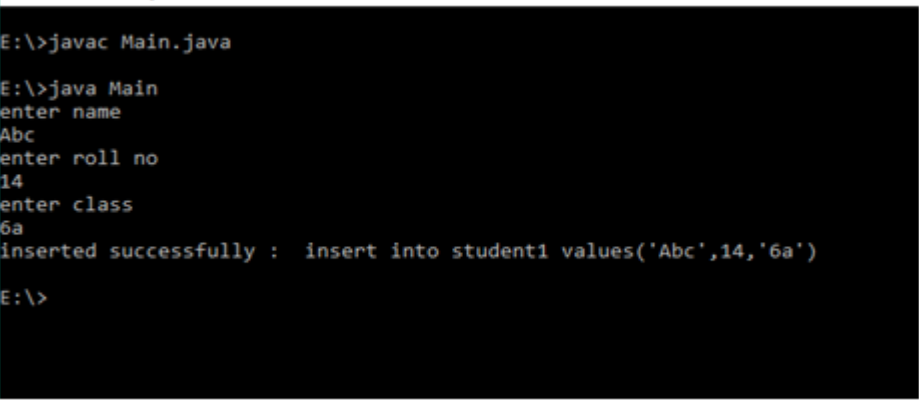
//Inserting data using SQL query
String sql = "insert into student1 values('"+name+"','"+roll+"','"+cls+"')";
Connection con=null;
try
{
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    //Reference to connection interface
    con = DriverManager.getConnection(url,user,pass);

    Statement st = con.createStatement();
    int m = st.executeUpdate(sql);
    if (m == 1)
        System.out.println("inserted successfully : "+sql);
    else
        System.out.println("insertion failed");
    con.close();
}
catch(Exception ex)
{
    System.err.println(ex);
}
}
}

```

OUTPUT



```

C:\Windows\System32\cmd.exe
E:\>javac Main.java
E:\>java Main
enter name
Abc
enter roll no
14
enter class
6a
inserted successfully : insert into student1 values('Abc',14,'6a')
E:\>

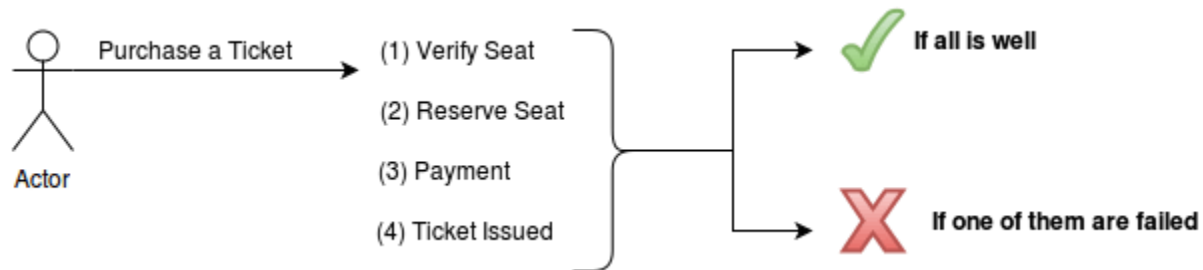
```

Transaction Management in JDBC

A transaction represents a group of operations, used to perform a task. A transaction is a set of commands; it will take our database from one consistent state to another consistent state.

The most important points about Transaction Management in JDBC are as follows:

- A transaction means, it is a group of operations used to perform a task.
- A transaction can reach either success state or failure state.
- If all operations are completed successfully then the transaction becomes success.
- If any one of the operation fail then all remaining operations will be cancelled and finally transaction will reach to fail state.



Types of Transactions:

The basic transactions are of two types.

- Local Transactions
- Global / Distributed Transactions

Local Transactions: - If all the operations are executed on one/same database, then it is called as local transaction.

Global / Distributed Transaction: - If the operations are executed on more than one database then it is called as global transactions.

Example: If we transfer the money from account1 to account2 of same bank, then it is called as local transaction. If we transfer the money from account1 to account2 of different banks, then it is called as global or distributed transaction.

- JDBC can support only local transactions. For distributed transactions, we must use either EJB technology or Spring Framework.

Transaction Management in JDBC Example:

We can get the Transaction support in JDBC from Connection interface. The Connection interface has 3 methods to perform Transaction Management in JDBC.

- `setAutoCommit()`
- `commit()`

- `rollback()`

Transaction setAutoCommit() :

Before going to begin the operations, first we need to disable the auto commit mode. This can be done by calling **`setAutoCommit(false)`**.

By default, all operations done from the java program are going to execute permanently in database. Once the permanent execution happened in database, we can't revert back them (Transaction Management is not possible).

Transaction commit() :

If all operations are executed successfully, then we commit a transaction manually by calling the **`commit()`** method.

Transaction rollback() :

If any one of the operation failed, then we cancel the transaction by calling **`rollback()`** method.

```
connection.setAutoCommit(false);
```

```
try
{
    -----
    -----

    connection.commit();
}
catch(Exception e)
{
    connection.rollback();
}
```