|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr.**  **No.** | **Date** | **Name of the Experiment** | **Page**  **No.** | **Sign** | **Remark** |
|  | 17/2/2023 | Write a program to demonstrate status of key on an Applet window such as KeyPressed, KeyReleased, KeyUp, KeyDown. |  |  |  |
|  | 24/2/2023 | Write a program to create a frame using AWT. Implement mouseClicked, mouseEntered() and mouseExited() events. Frame should become visible when the mouse enters it. |  |  |  |
|  | 3/3/2023 | Develop a GUI which accepts the information regarding the marks for all the subjects of a student in the examination. Display the result for a student in a separate window. |  |  |  |
|  | 11/3/2023 | Write a program to insert and retrieve the data from the database using JDBC. |  |  |  |
|  | 17/3/2023 | Develop an RMI application which accepts a string or a number and checks that string or number is palindrome or not. |  |  |  |
|  | 24/3/2023 | Write a program to demonstrate the use of InetAddress class and its factory methods. |  |  |  |
|  | 21/4/2023 | A. Write Servlet (procedure for client side) to display the username and password accepted from the client.  B. Write Servlet (procedure for server side) to display the username and password accepted from the client. |  |  |  |
|  | 28/4/2023 | Write a database application that uses any JDBC driver. |  |  |  |
|  | 5/5/2023 | Write a simple JSP page to display a simple message (It may be a simple html page). |  |  |  |
|  | 23/5/2023 | Create a simple calculator application using servlet. |  |  |  |

**Index**

**EXPERIMENT NO. 01**

**Aim:** Write a program to demonstrate status of key on an Applet window such as KeyPressed, KeyReleased, KeyUp, KeyDown.

**Objective:** The objective of this assignment is to learn the concepts of Keyboard event handling using Applet.

**Outcome:**

The practical is expected to develop the following skill:

1. Able to implement the events of KeyListerner on an Applet window.

**Theory:**

An event can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard or page scrolling, etc.

The java.awt.event package can be used to provide various event classes.

**Classification of Events:**

**1. Foreground Events**

Foreground events are the events that require user interaction to generate, i.e., foreground events are generated due to interaction by the user on components in Graphic User Interface (**GUI**). Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor moments, etc.

**2. Background Events**

Events that don’t require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, hardware or software failure, timer expires, operation completion, etc.

**Event Handling:**

It is a mechanism to **control the events** and to **decide 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. To handle the events, Java follows the ***Delegation Event model.***

**Delegation Event model:**

1. **Source:** The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides classes for source object. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.
2. **Listeners:** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events. 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 listener processes the event and then returns.

To perform Event Handling, we need to register the source with the listener.

**Registering the Source with Listener**

Different Classes provide different registration methods.

**Syntax:**

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener.

**Example 1:** For **KeyEvent** we use *addKeyListener()* to register.

**Example 2:** that For **ActionEvent** we use *addActionListener()* to register.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

A **KeyEvent** is generated when keyboard input occurs. Whenever a user presses any key on the Keyboard, different events are fired. There are three types of key events,  
which are identified by these integer constants: **KEY\_PRESSED**, **KEY\_RELEASED**, and  
**KEY\_TYPED**. The first two events are generated when any key is pressed or released. The  
last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered. Their general forms are shown here:

**char getKeyChar( )**

|  |  |  |  |
| --- | --- | --- | --- |
| **Event Class** | **Listener Interface** | **Methods** | **Description** |
| KeyEvent | KeyListener | keyTyped ( ) | Called just after the user types a Unicode character into the listened-to component. |
| keyPressed ( ) | Called just after the user presses a key while the listened-to component has the focus. |
| keyReleased ( ) | Called just after the user releases a key while the listened-to component has the focus. |

**Applet:**

An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included in a page. When you use a Java technology-enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM). The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet  
more conveniently, simply include a comment at the head of your Java source code file that  
contains the APPLET tag. Here is an example of such a comment:

/\*  
<applet code="MyApplet" width=200 height=60>

</applet>  
\*/  
This comment contains an APPLET tag that will run an applet called **MyApplet** in a  
window that is 200 pixels wide and 60 pixels high.

**Applet Life Cycle:**

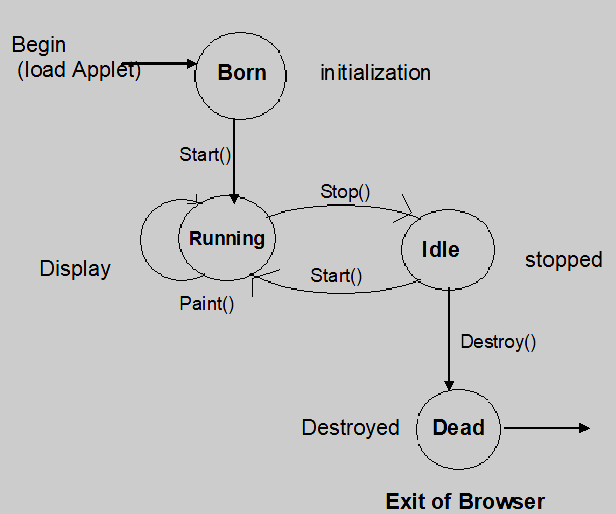


Figure 1: Applet state transition diagram

**Initialization state:**

Applet enters the initialization state when it is first loaded. This is achieved by calling the init() method of Applet class.

* Create objects needed by the applet.
* Set up initial values.
* Load images or fonts.
* Set up colors.

**Running state:**

Applet enters the running state when the system calls the start() method of Applet class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in “stopped” (idle) state.

**Idle or Stopped State:**

An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can calling the stop () method explicitly.

**Dead State:**

An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the destroy() method when we quit the browser.

**Display State:**

Applet moves to the display state whenever it has to perform some output operation on the screen. This happens immediately after the applet enters into the running state.

**Simple Applet Display Methods:**

To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class. Typically, it is called from within either **update( )** or **paint( )**. It has the following general form:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x,y.* In a Java window, the upper-left corner is location 0,0.

In addition to displaying information in its window, an applet can also output a message  
to the status window of the browser or applet viewer on which it is running. To do so, call  
showStatus( ) with the string that you want displayed. The status window is a good place  
to give the user feedback about what is occurring in the applet, suggest options, or possibly  
report some types of errors.

**Conclusion:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPERIMENT NO. 02**

**Aim:** Write a program to create a frame using AWT. Implement mouseClicked, mouseEntered() and mouseExited() events. Frame should become visible when the mouse enters it.

**Objective:** The objective of this assignment is to learn the concepts of mouse event handling using AWT.

**Outcome:**

The practical is expected to develop the following skill:

1. Able to implement the events of MouseListerner using AWT.

**THEORY** **:**

**Theory:**

**Java AWT** (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The AWT defines windows according to a class hierarchy that adds functionality and  
specificity with each level. The two most common windows are those derived from  
**Panel**, which is used by applets, and those derived from **Frame**, which creates a standard  
application window.

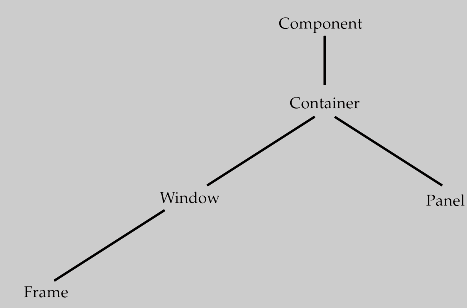


Figure 1: The class hierarchy for Panel and Frame

**Component**  
At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that  
encapsulates all of the attributes of a visual component. All user interface elements that are  
displayed on the screen and that interact with the user are subclasses of **Component**. It defines  
over a hundred public methods that are responsible for managing events, such as mouse and  
keyboard input, positioning and sizing the window, and repainting.

**Container**The **Container** class is a subclass of **Component**. It has additional methods that allow other  
**Component** objects to be nested within it. Other **Container** objects can be stored inside of a  
**Container** (since they are themselves instances of **Component**). This makes for a multileveled  
containment system. A container is responsible for laying out (that is, positioning) any components that it contains.

**Panel**

The **Panel** class is a concrete subclass of **Container**. It doesn’t add any new methods; it simply  
implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen  
component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet,  
it is drawn on the surface of a **Panel** object.

Other components can be added to a **Panel** object by its **add( )** method (inherited from  
**Container**). Once these components have been added, you can position and resize them  
manually using the **setLocation( )**, **setSize( )**, **setPreferredSize( )**, or **setBounds( )** methods  
defined by **Component**.

**Window**  
The **Window** class creates a top-level window. A *top-level window* is not contained within any  
other object; it sits directly on the desktop. Generally, you won’t create **Window** objects  
directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

**Frame**  
**Frame** encapsulates what is commonly thought of as a “window.” It is a subclass of **Window**and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from  
within an applet, it will contain a warning message, such as “Java Applet Window,” to the  
user that an applet window has been created. When a **Frame** window is created  
by a stand-alone application rather than an applet, a normal window is created.

**Working with Frame Windows**

After the applet, the type of window you will most often create is derived from **Frame**. You will  
use it to create child windows within applets, and top-level or child windows for stand-alone  
applications. As mentioned, it creates a standard-style window. Here are two of **Frame**’s constructors:

Frame( )

Frame(String *title*)

The first form creates a standard window that does not contain a title. The second form creates  
a window with the title specified by *title.* Notice that you cannot specify the dimensions of  
the window. Instead, you must set the size of the window after it has been created.

There are several key methods you will use when working with **Frame** windows. They  
are examined here.

**Setting the Window’s Dimensions**

The **setSize( )** method is used to set the dimensions of the window. Its signature is shown here:

void setSize(int *newWidth*, int *newHeight*)

void setSize(Dimension *newSize*)

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and  
**height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in  
terms of pixels.

The **getSize( )** method is used to obtain the current size of a window. Its signature is  
shown here:

Dimension getSize( )

This method returns the current size of the window contained within the **width** and **height**fields of a **Dimension** object.

**Hiding and Showing a Window**

After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:

void setVisible(boolean *visibleFlag*)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

**Setting a Window’s Title**

You can change the title in a frame window using **setTitle( )**, which has this general form:

void setTitle(String *newTitle*)

Here, *newTitle* is the new title for the window.

**Closing a Frame Window**

When using a frame window, your program must remove that window from the screen when  
it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement  
the **windowClosing( )** method of the **WindowListener** interface. Inside **windowClosing( )**,  
you must remove the window from the screen. The example in the next section illustrates  
this technique.

**Handling Events in a Frame Window**

Since Frame is a subclass of Component, it inherits all the capabilities defined by Component.  
This means that you can use and manage a frame window just like you manage an applet’s  
main window. For example, you can override paint( ) to display output, call repaint( ) when  
you need to restore the window, and add event handlers. Whenever an event occurs in a  
window, the event handlers defined by that window will be called. Each window handles its  
own events. For example, the following program creates a window that responds to mouse  
events.

The java.awt.event package can be used to provide various event classes.

**Classification of Events:**

**1. Foreground Events**

Foreground events are the events that require user interaction to generate, i.e., foreground events are generated due to interaction by the user on components in Graphic User Interface (**GUI**). Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor moments, etc.

**2. Background Events**

Events that don’t require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, hardware or software failure, timer expires, operation completion, etc.

**Event Handling:**

It is a mechanism to **control the events** and to **decide 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. To handle the events, Java follows the ***Delegation Event model.***

**Delegation Event model:**

1. **Source:** The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides classes for source object. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.
2. **Listeners:** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events. 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 listener processes the event and then returns.

To perform Event Handling, we need to register the source with the listener.

**Registering the Source with Listener**

Different Classes provide different registration methods.

**Syntax:**

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener.

**MouseEvent class**

This event indicates a mouse action occurred in a component. This low-level event is generated by a component object for Mouse Events and Mouse motion events.

**Constructor:**

**MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger)**

Constructs a MouseEvent object with the specified:

**source**- source component,

**id**- type of event,

**when**- system time at mouse event occurred

**modifiers**-to know what modifiers were pressed after event was occurred,

**x & y**- coordinates of the mouse ,

**clickCount**- click count

**popupTrigger**- whether popup menu appeared

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

int getX( )

int getY( )

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.  
The **translatePoint( )** method changes the location of the event. Its form is shown here:

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.  
The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this  
platform. Its form is shown here:

boolean isPopupTrigger( )

Also available is the **getButton( )** method, shown here:

int getButton( )

It returns a value that represents the button that caused the event. The return value will be  
one of these constants defined by **MouseEvent**:



The **NOBUTTON** value indicates that no button was pressed or released.  
Java SE 6 added three methods to **MouseEvent** that obtain the coordinates of the mouse  
relative to the screen rather than the component.

They are shown here:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and  
Y coordinate. The other two methods return the indicated coordinate.

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| int getClickCount() | Returns the number of quick, consecutive clicks the user has made (including this event). For example, returns 2 for a double click. |
| int getButton() | Returns which mouse button, if any, has a changed state. One of the following constants is returned: NOBUTTON, BUTTON1, BUTTON2, or BUTTON3. |
| Int getX() int getY() | Return the (x,y) position at which the event occurred, relative to the component that fired the event. |
| Point getPoint() | Returns the x,y position of the event rlative to the source component. |

**MouseListener Interface:**

* Mouse events notify when the user uses the mouse (or similar input device) to interact with a component.
* Mouse events occur when the cursor enters or exits a component's onscreen area and when the user presses or releases one of the mouse buttons.

|  |  |  |  |
| --- | --- | --- | --- |
| **Event Class** | **Listener Interface** | **Methods** | **Description** |
| MouseEvent | MouseListener | mouseClicked ( ) | Called just after the user clicks the listened-to component. |
| mouseEntered ( ) | Called just after the cursor enters the bounds of the listened-to component. |
| mouseExited ( ) | Called just after the cursor exits the bounds of the listened-to component. |
| mousePressed ( ) | Called just after the user presses a mouse button while the cursor is over the listened-to component. |
| mouseReleased() | Called just after the user releases a mouse button after a mouse press over the listened-to component. |

**CONCLUSION :**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**EXPERIMENT NO. 03**

**TITLE:** Developing GUI to accepts information of students.

**AIM:** Develop a GUI which accepts the information regarding the marks for all the subjects of a student in the examination. Display the result for a student in a separate window.

**THEORY**:

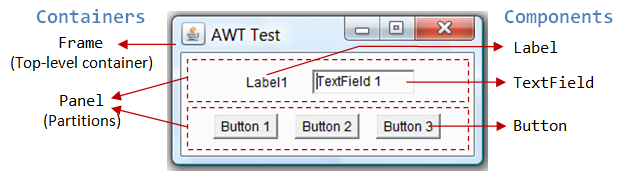
**Programming GUI with AWT**

AWT is huge! It consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 8). Fortunately, only 2 packages - java.awt and java.awt.event - are commonly-used.

* The java.awt package contains the core AWT graphics classes:
* GUI Component classes, such as Button, TextField, and Label.
* GUI Container classes, such as Frame and Panel.
* Layout managers, such as FlowLayout, BorderLayout and GridLayout.
* Custom graphics classes, such as Graphics, Color and Font.
* The java.awt.event package supports event handling:
* Event classes, such as ActionEvent, MouseEvent, KeyEvent and WindowEvent,
* Event Listener Interfaces, such as ActionListener, MouseListener, MouseMotionListener, KeyListener and WindowListener,
* Event Listener Adapter classes, such as MouseAdapter, KeyAdapter, and WindowAdapter.

AWT provides a platform-independent and device-independent interface to develop graphic programs that runs on all platforms, including Windows, macOS, and Unixes.

**AWT Containers and Components :**



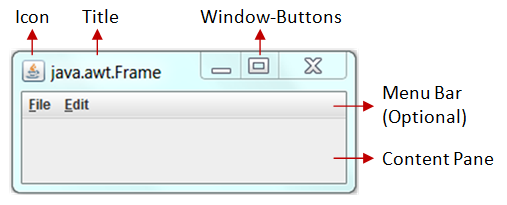
There are two groups of GUI elements:

* Component (Widget, Control): Components are elementary GUI entities, such as Button, Label, and TextField. They are also called widgets, controls in other graphics systems.
* Container: Containers, such as Frame and Panel, are used to hold components in a specific layout (such as FlowLayout or GridLayout). A container can also hold sub-containers.

**AWT Container Classes :**

Top-Level Containers: Frame, Dialog and Applet

Each GUI program has a *top-level container*. The commonly-used top-level containers in AWT are Frame, Dialog and Applet:



A Frame provides the "main window" for your GUI application. It has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area.

**CONCLUSION :**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**EXPERIMENT NO. 04**

**Aim:** Study the concept of Java Database Connectivity concept to write a suitable application.

**Objective:** To insert and retrieve data from MySQL database table using SQL Query commands.

**Outcome:**

The practical is expected to develop the following skill:

1. Able to insert and retrieve data from MySQL database table.

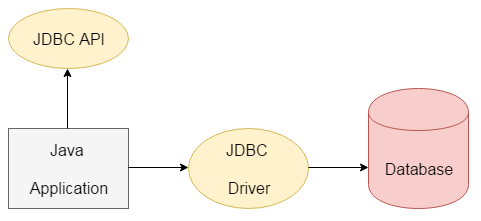
**Theory:**

* **The Concept of JDBC:**

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

* JDBC-ODBC Bridge Driver,
* Native Driver,
* Network Protocol Driver, and
* Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

* Driver interface
* Connection interface
* Statement interface
* PreparedStatement interface
* CallableStatement interface
* ResultSet interface
* ResultSetMetaData interface
* DatabaseMetaData interface
* RowSet interface

A list of popular *classes* of JDBC API are given below:

* DriverManager class
* Blob class
* Clob class
* Types class
* **Why Should We Use JDBC**

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

## What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

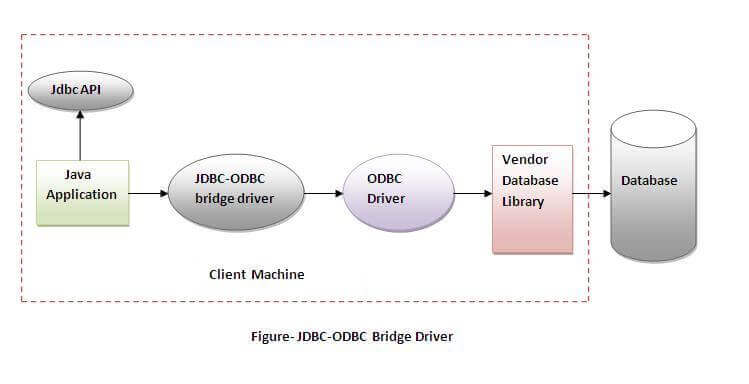
* **JDBC Driver Types & Architecture:**

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

### 1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.



Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

**Advantages**:

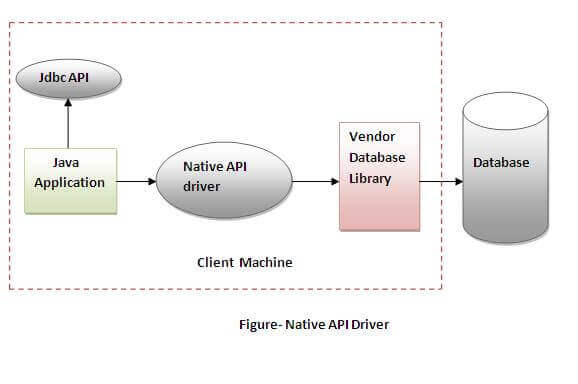
* easy to use.
* can be easily connected to any database.

**Disadvantages**:

* Performance degraded because JDBC method call is converted into the ODBC function calls.
* The ODBC driver needs to be installed on the client machine.

### 2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



**Advantage**:

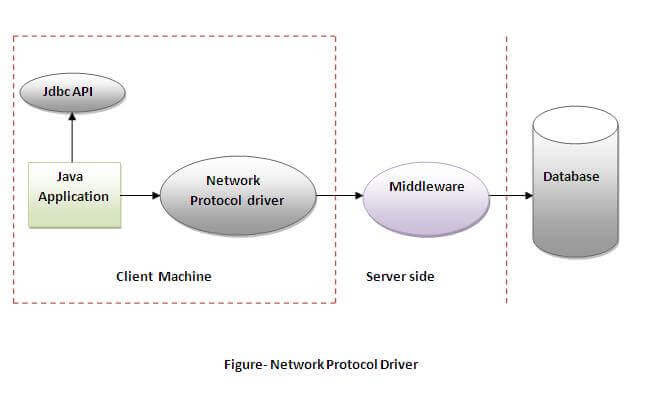
* performance upgraded than JDBC-ODBC bridge driver.

**Disadvantage**:

* The Native driver needs to be installed on the each client machine.
* The Vendor client library needs to be installed on client machine.

### 3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



**Advantage**:

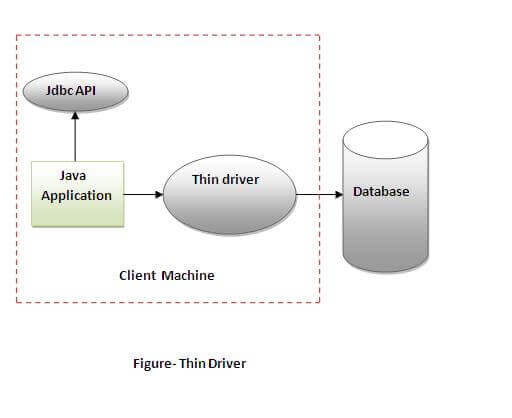
* No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

**Disadvantages**:

* Network support is required on client machine.
* Requires database-specific coding to be done in the middle tier.
* Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

### 4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.



**Advantage**:

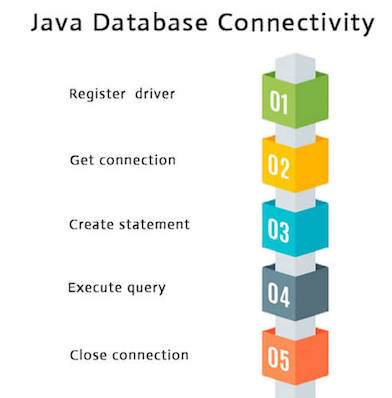
* Better performance than all other drivers.
* No software is required at client side or server side.

**Disadvantage**:

* Drivers depend on the Database

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the Driver class
* Create connection
* Create statement
* Execute queries
* Close connection



**CONCLUSION :**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**EXPERIMENT NO. 05**

**Aim:** Develop an RMI application which accepts a string or a number and checks that string or number is palindrome or not

**Objective:** The objective of this practical is to use Remote Method Invocation.

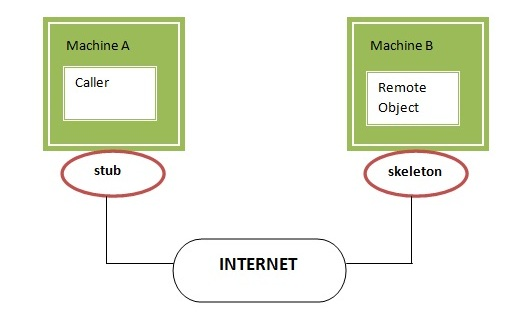
**Outcome:**

The practical is expected to develop the following skill:

1. Students are able to develop their own remote interface, and able to implement RMI server, and RMI client.

**Theory:**

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM. RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java. rmi.



Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

### stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

### skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons

## Java RMI Example

This is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

**ALGORITHM:**  
  
**SERVER SIDE:**  
Step 1: Start  
Step 2: Define the class rmiserver  
Step 3: Create the object twox in try  
Step 4: Register the object twox  
Step 5: Display the exception in catch  
Step 6: Stop  
  
**CLIENT SIDE:**  
Step 1: Start  
Step 2: Define the class rmiclient  
Step 3: Initialize the string s1 in try  
Step 4: Create and Initialize the object onex  
Step 5: Assign the value to m by calling the method palin  
Step 6: Check whether the string is palindrome or not  
Step 7: Display whether the string is palindrome or not  
Step 8: Display the exception in catch  
Step 9: Stop

**CONCLUSION:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPERIMENT NO. 06**

**AIM:** Write a program to demonstrate the use of InetAddress class and its factory methods.

**THEORY:**

**java.net.InetAddress Class in Java :**

**public class InetAddress**extends **Object** implements **Serializable:**

The **java.net.InetAddress** class provides methods to get the IP address of any hostname. An IP address is represented by 32-bit or 128-bit unsigned number. InetAddresscan handle both IPv4 and IPv6 addresses.

There are 2 types of addresses :

Unicast — An identifier for a single interface.

Multicast — An identifier for a set of interfaces.

**InetAddress – Factory Methods :**

The InetAddress class is used to encapsulate both, the numerical IP address and the domain name for that address. The InetAddress class has no visible constructors. The InetAddress class has the inability to create objects directly, hence factory methods are used for the purpose. Factory Methods are static methods in a class that return an object of that class.

There are 5 factory methods available in InetAddress class –

| Method | Description |
| --- | --- |
| static InetAddress getLocalHost() throws UnknownHostException | This method returns the instance of InetAddress containing the local hostname and address. |
| public static InetAddress getByName( String host ) throws UnknownHostException | This method returns the instance of InetAddress containing LocalHost IP and name. |
| static InetAddress[] getAllByName( String hostName ) throws UnknownHostException | This method returns the array of the instance of InetAddress class which contains IP addresses. |
| static InetAddress getByAddress( byte IPAddress[] ) throws UnknownHostException | This method returns an InetAddress object created from the raw IP address. |
| static InetAddress getByAddress( String hostName, byte IPAddress[] ) throws UnknownHostException | This method creates and returns an InetAddress based on the provided hostname and IP address. |

**Conclusion:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPERIMENT NO. 07**

**Aim:** Understand the concept of Servlet, Client and Server Architecture using Java Programming.

**Objective:** Write Servlet (procedure for client and Server side) to display the username and password accepted from the client.

**Outcome:**

The practical is expected to develop the following skill:

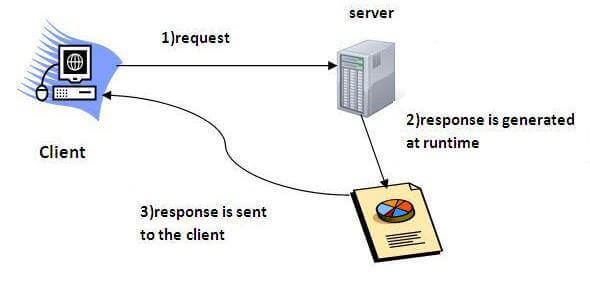
1. Students will be able to develop Servlet based Client and Server Architecture using Java Programming.

**Theory:**

**Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).

**Servlet** technology is robust and scalable because of java language.

* Servlet is a technology which is used to create a web application.
* Servlet is an API that provides many interfaces and classes including documentation.
* Servlet is an interface that must be implemented for creating any Servlet.
* Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
* Servlet is a web component that is deployed on the server to create a dynamic web page.



### Advantages of Servlet:

There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) manages Servlets, so we don't need to worry about the memory leak, [garbage collection](https://www.javatpoint.com/Garbage-Collection), etc.
4. **Secure:** because it uses java language.

# Web Terminology

|  |  |
| --- | --- |
| **Servlet Terminology** | **Description** |
| [Website: static vs dynamic](https://www.javatpoint.com/website-static-vs-dynamic) | It is a collection of related web pages that may contain text, images, audio and video. |
| [HTTP](https://www.javatpoint.com/http) | It is the data communication protocol used to establish communication between client and server. |
| [HTTP Requests](https://www.javatpoint.com/http-requests) | It is the request send by the computer to a web server that contains all sorts of potentially interesting information. |
| [Get vs Post](https://www.javatpoint.com/get-vs-post) | It gives the difference between GET and POST request. |
| [Container](https://www.javatpoint.com/container) | It is used in java for dynamically generating the web pages on the server side. |
| [Server: Web vs Application](https://www.javatpoint.com/server-web-vs-application) | It is used to manage the network resources and for running the program or software that provides services. |
| [Content Type](https://www.javatpoint.com/content-type) | It is HTTP header that provides the description about what are you sending to the browser. |

**Conclusion:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPERIMENT NO. 08**

**Aim:** Study the concept of Java Database Connectivity and its drivers concept to write a suitable application.

**Objective:** Write a database application that uses any JDBC driver to insert and retrieve data from MySQL database table using SQL Query commands.

**Outcome:**

The practical is expected to develop the following skill:

1. Able to insert and retrieve data from MySQL database table.

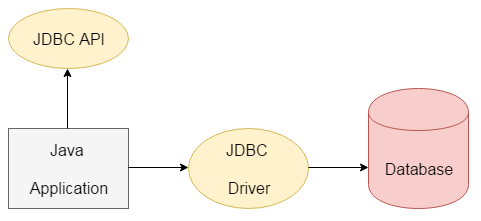
**Theory:**

* **The Concept of JDBC:**

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

* JDBC-ODBC Bridge Driver,
* Native Driver,
* Network Protocol Driver, and
* Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

* Driver interface
* Connection interface
* Statement interface
* PreparedStatement interface
* CallableStatement interface
* ResultSet interface
* ResultSetMetaData interface
* DatabaseMetaData interface
* RowSet interface

A list of popular *classes* of JDBC API are given below:

* DriverManager class
* Blob class
* Clob class
* Types class
* **Why Should We Use JDBC**

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

## What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

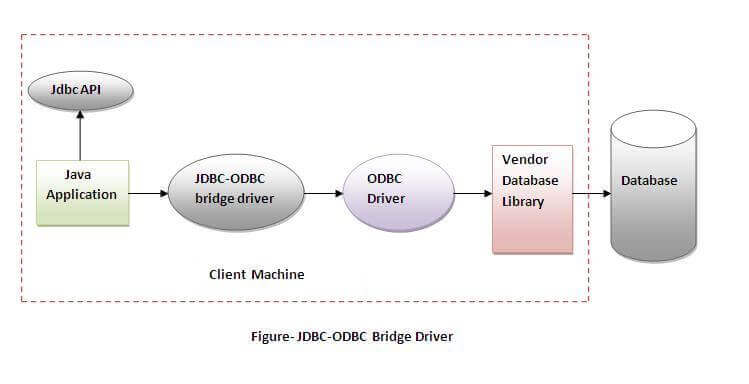
* **JDBC Driver Types & Architecture:**

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

### 1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.



Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

**Advantages**:

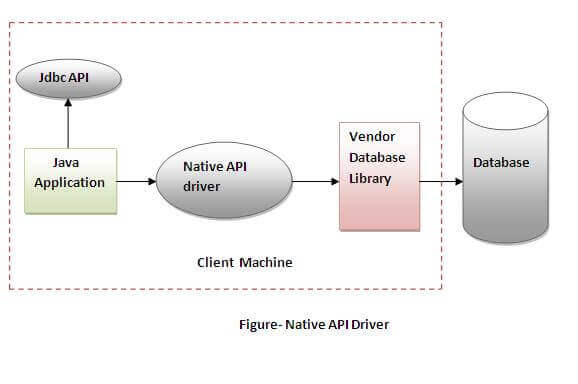
* easy to use.
* can be easily connected to any database.

**Disadvantages**:

* Performance degraded because JDBC method call is converted into the ODBC function calls.
* The ODBC driver needs to be installed on the client machine.

### 2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



**Advantage**:

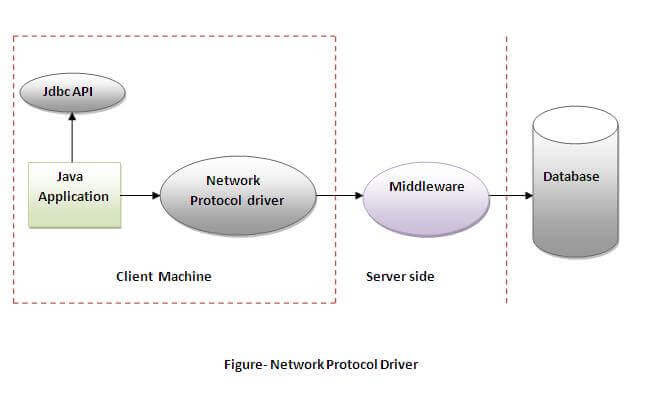
* performance upgraded than JDBC-ODBC bridge driver.

**Disadvantage**:

* The Native driver needs to be installed on the each client machine.
* The Vendor client library needs to be installed on client machine.

### 3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



**Advantage**:

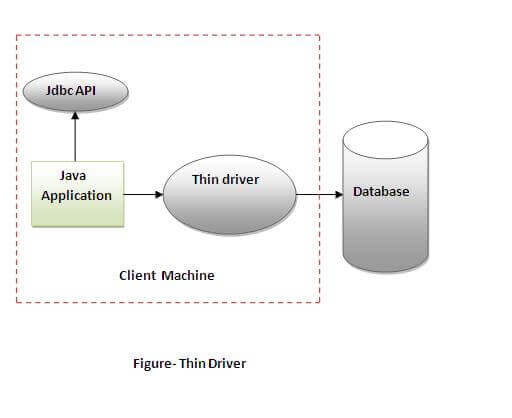
* No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

**Disadvantages**:

* Network support is required on client machine.
* Requires database-specific coding to be done in the middle tier.
* Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

### 4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.



**Advantage**:

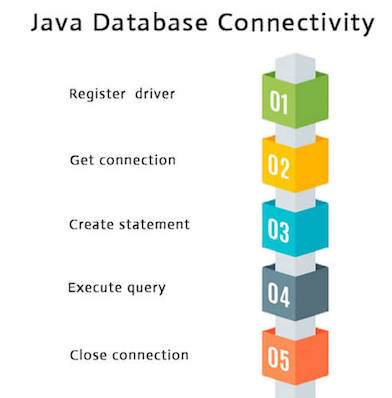
* Better performance than all other drivers.
* No software is required at client side or server side.

**Disadvantage**:

* Drivers depend on the Database

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the Driver class
* Create connection
* Create statement
* Execute queries
* Close connection



**Conclusion:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPERIMENT NO. 09**

**AIM:** Write a simple JSP page to display a simple message (It may be a simple html page).

**OBJECTIVE:** Create JSP page to receive user email ID and print it on local host.

**OUTCOME:**

The practical is expected to develop the following skill:

1. Able to create JSP pages using java programming.
2. Able to use Apache Tomcat Server.

**THEORY:**

JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc.

A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags, etc.

### Advantages of JSP over Servlet

There are many advantages of JSP over the Servlet. They are as follows:

#### 1) Extension to Servlet

JSP technology is the extension to Servlet technology. We can use all the features of the Servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

#### 2) Easy to maintain

JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

#### 3) Fast Development: No need to recompile and redeploy

If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

#### 4) Less code than Servlet

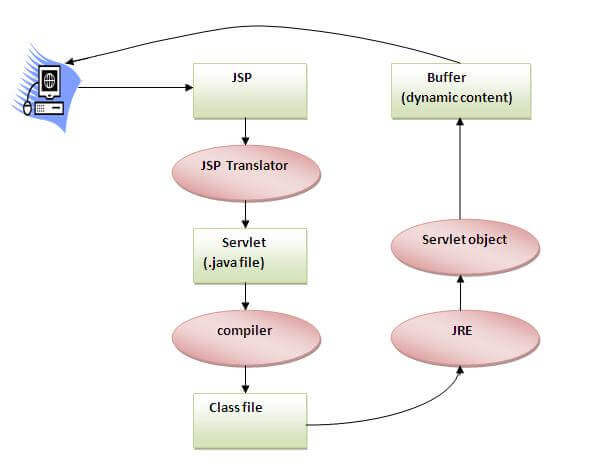
In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

**The Lifecycle of a JSP Page**

The JSP pages follow these phases:

* Translation of JSP Page
* Compilation of JSP Page
* Classloading (the classloader loads class file)
* Instantiation (Object of the Generated Servlet is created).
* Initialization ( the container invokes jspInit() method).
* Request processing ( the container invokes \_jspService() method).
* Destroy ( the container invokes jspDestroy() method).

As depicted in the above diagram, JSP page is translated into Servlet by the help of JSP translator. The JSP translator is a part of the web server which is responsible for translating the JSP page into Servlet. After that, Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happen in Servlet are performed on JSP later like initialization, committing response to the browser and destroy.



**Conclusion:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPERIMENT NO. 10**

**AIM:**Write a java program to create a simple calculator application using servlet.

**OBJECTIVE:** Create a simple calculator application using servlet to perform operations such as addition, subtraction, multiplication and division.

**OUTCOME:**

The practical is expected to develop the following skill:

* Able to create Servlet using java programming.
* Able to design GUI for calculation application.

**THEORY:**

**Servlet interface:**

**Servlet interface provides** commonbehaviorto all the servlets.Servlet interface defines methods that all servlets must implement.

Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

**Methods of Servlet interface:**

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **public void init(ServletConfig config)** | initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once. |
| **public void service(ServletRequest request,ServletResponse response)** | provides response for the incoming request. It is invoked at each request by the web container. |
| **public void destroy()** | is invoked only once and indicates that servlet is being destroyed. |
| **public ServletConfig getServletConfig()** | returns the object of ServletConfig. |
| **public String getServletInfo()** | returns information about servlet such as writer, copyright, version etc. |

**GenericServlet class:**

**GenericServlet** class implements **Servlet**, **ServletConfig** and **Serializable** interfaces. It provides the implementation of all the methods of these interfaces except the service method.

GenericServlet class can handle any type of request so it is protocol-independent.

You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

1. Methods of GenericServlet class:
2. **public void init(ServletConfig config)** is used to initialize the servlet.
3. **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
4. **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
5. **public ServletConfig getServletConfig()** returns the object of ServletConfig.
6. **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
7. **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)
8. **public ServletContext getServletContext()** returns the object of ServletContext.
9. **public String getInitParameter(String name)** returns the parameter value for the given parameter name.
10. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
11. **public String getServletName()** returns the name of the servlet object.
12. **public void log(String msg)** writes the given message in the servlet log file.
13. **public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace

**HttpServlet class:**

The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

**Methods of HttpServlet class:**

1. **public void service(ServletRequest req,ServletResponse res)** dispatches the request to the protected service method by converting the request and response object into http type.
2. **protected void service(HttpServletRequest req, HttpServletResponse res)** receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.
3. **protected void doGet(HttpServletRequest req, HttpServletResponse res)** handles the GET request. It is invoked by the web container.
4. **protected void doPost(HttpServletRequest req, HttpServletResponse res)** handles the POST request. It is invoked by the web container.
5. **protected void doHead(HttpServletRequest req, HttpServletResponse res)** handles the HEAD request. It is invoked by the web container.
6. **protected void doOptions(HttpServletRequest req, HttpServletResponse res)** handles the OPTIONS request. It is invoked by the web container.
7. **protected void doPut(HttpServletRequest req, HttpServletResponse res)** handles the PUT request. It is invoked by the web container.
8. **protected void doTrace(HttpServletRequest req, HttpServletResponse res)** handles the TRACE request. It is invoked by the web container.
9. **protected void doDelete(HttpServletRequest req, HttpServletResponse res)** handles the DELETE request. It is invoked by the web container.
10. **protected long getLastModified(HttpServletRequest req)** returns the time when HttpServletRequest was last modified since midnight January 1, 1970 GMT.

**CONCLUSION:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_