JFrame (javax.swing.JFrame)

Method/Property	Data Type	Description
setTitle(String title)	void	Sets the title of the frame.
getTitle()	String	Returns the title of the frame.
setSize(int width, int height)	void	Sets the size of the frame.
setBounds(int x, int y, int width, int height)	void	Sets the position and size of the frame.
setDefaultCloseOperation(int operation)	void	Defines the operation when the frame is closed.
setVisible(boolean b)	void	Controls visibility of the frame.
setResizable(boolean resizable)	void	Enables/disables frame resizing.
setLayout(LayoutManager mgr)	void	Sets the layout manager.
add(Component comp)	Component	Adds a component to the frame.
remove(Component comp)	void	Removes a component from the frame.
getContentPane()	Container	Returns the content pane of the frame.
pack()	void	Sizes the frame to fit its components.
setExtendedState(int state)	void	Maximizes/minimizes the frame.

.JPanel (javax.swing.JPanel)

Method/Property	Data Type	Description
setLayout(LayoutManager mgr)	void	Sets the layout manager.
add(Component comp)	Component	Adds a component to the panel.
remove(Component comp)	void	Removes a component from the panel.
setBackground(Color c)	void	Sets the background color.
setBorder(Border border)	void	Sets a border for the panel.

JLabel (javax.swing.JLabel)

Method/Property	Data Type	Description
setText(String text)	void	Sets the label text.
getText()	String	Returns the text of the label.
setIcon(Icon icon)	void	Sets an icon on the label.
setHorizontalAlignment(int alignment)	void	Sets horizontal text alignment.
setVerticalAlignment(int alignment)	void	Sets vertical text alignment.

JTextField (javax.swing.JTextField)

Method/Property	Data Type	Description
setText(String text)	void	Sets the text in the text field.
getText()	String	Retrieves the text from the text field.
setEditable(boolean b)	void	Enables/disables user input.
setColumns(int columns)	void	Sets the number of columns.

JButton (javax.swing.JButton)

Method/Property	Data Type	Description
setText(String text)	void	Sets the button text.
getText()	String	Retrieves the button text.
addActionListener(ActionListener l)	void	Adds an action listener.
setEnabled(boolean b)	void	Enables/disables the button.
setIcon(Icon icon)	void	Sets an icon for the button.

JCheckBox (javax.swing.JCheckBox)

Method/Property	Data Type	Description
setSelected(boolean b)	void	Sets the checkbox state.
isSelected()	boolean	Checks if the checkbox is selected.

JRadioButton (javax.swing.JRadioButton)

Method/Property	Data Type	Description
setSelected(boolean b)	void	Selects/deselects the radio button.
isSelected()	boolean	Returns the selection state.

JComboBox (javax.swing.JComboBox)

Method/Property	Data Type	Description
addItem(Object item)	void	Adds an item to the combo box.
removeItem(Object item)	void	Removes an item from the combo box.
setSelectedIndex(int index)	void	Selects an item by index.
getSelectedItem()	Object	Returns the selected item.

JTextArea (javax.swing.JTextArea)

Method/Property	Data Type	Description
setText(String text)	void	Sets the text in the text area.
getText()	String	Retrieves the text.
setRows(int rows)	void	Sets the number of rows.
setColumns(int columns)	void	Sets the number of columns.

..JWindow

Method/Property	Data Type	Description
setVisible(boolean b)	void	Sets the visibility of the window.
getOwner()	Window	Returns the owner of this window.
setLocation(int x, int y)	void	Sets the location of the window.
setSize(int width, int height)	void	Sets the size of the window.
getContentPane()	Container	Returns the content pane of the window.
add(Component c)	Component	Adds a component to the window.
dispose()	void	Releases the resources used by the window.
pack()	void	Adjusts the window size to fit its components.

.JApplet

Method/Property	Data Type	Description
init()	void	Called for applet initialization.
start()	void	Called when the applet is started.
stop()	void	Called when the applet is stopped.
destroy()	void	Called when the applet is destroyed.
getContentPane()	Container	Returns the content pane of the applet.
setJMenuBar(JMenuBar bar)	void	Sets the menu bar for the applet.

.JPasswordField

Method/Property	Data Type	Description
setEchoChar(char c)	void	Sets the character to display instead of actual text.
getPassword()	char[]	Returns the password entered.
setText(String text)	void	Sets the text in the password field.
getText()	String	Returns the text in the field (deprecated).

.JToolTip

Method/Property	Data Type	Description
setTipText(String text)	void	Sets the tooltip text.
getTipText()	String	Returns the tooltip text.

.JTabbedPane

Method/Property	Data Type	Description
addTab(String title, Component component)	void	Adds a tab to the pane.
setSelectedIndex(int index)	void	Selects a tab by index.
getSelectedIndex()	int	Returns the selected tab index.
removeTabAt(int index)	void	Removes a tab at the specified index.

.JScrollPane

Method/Property	Data Type	Description
setViewportView(Component view)	void	Sets the component inside the scroll pane.
getVerticalScrollBar()	JScrollBar	Returns the vertical scroll bar.
getHorizontalScrollBar()	JScrollBar	Returns the horizontal scroll bar.

. .Menus & Menu Bars (JMenu, JMenuBar, JMenuItem)

Method/Property	Data Type	Description
add(JMenuItem item)	void	Adds a menu item to a menu.
addSeparator()	void	Adds a separator to the menu.
setMnemonic(char mnemonic)	void	Sets a keyboard shortcut.
getMenu(int index)	JMenu	Returns the menu at the given index.

.JOptionPane

Method/Property	Data Type	Description
showMessageDialog(Component parent, String message)	void	Displays an information message.
showConfirmDialog(Component parent, String message)	int	Displays a confirmation dialog.
showInputDialog(Component parent, String message)	String	Displays an input dialog.

.JDialog

Method/Property	Data Type	Description
setModal(boolean modal)	void	Sets whether the dialog blocks input to other windows.
setTitle(String title)	void	Sets the title of the dialog.
setSize(int width, int height)	void	Sets the size of the dialog.

.Event Handling & Delegation Model

Method/Property	Data Type	Description
addActionListener(ActionListener l)	void	Registers an action event listener.

addMouseListener(MouseListener l)	void	Registers a mouse event listener.
addKeyListener(KeyListener l)	void	Registers a keyboard event listener.

. vAdapter Classes

Class Name	Description
MouseAdapter	Provides empty implementations of MouseListener methods.
KeyAdapter	Provides empty implementations of KeyListener methods.
WindowAdapter	Provides empty implementations of WindowListener methods.

. JToolBar

Method/Property	Data Type	Description
add(Component c)	void	Adds a component to the toolbar.
setFloatable(boolean b)	void	Sets whether the toolbar can be dragged.
addSeparator()	void	Adds a separator to the toolbar.

. JColorChooser

Method/Property	Data Type	Description
showDialog(Component parent, String title, Color initialColor)	Color	Opens a color picker dialog.
getColor()	Color	Returns the selected color.

. Image Displaying Components

Method/Property	Data Type	Description
setIcon(Icon icon)	void	Sets an icon for JLabel, JButton, etc.
getIcon()	Icon	Returns the current icon.
setDisabledIcon(Icon icon)	void	Sets the icon when disabled.

. Basic Swing Components (JButton, JLabel, JTextField, JCheckBox, etc.)

Component	Method/Property	Data Type	Description
JButton	setText(String text)	void	Sets the button text.
JButton	getText()	String	Returns the button text.
JLabel	setText(String text)	void	Sets the label text.
JLabel	getText()	String	Returns the label text.
JTextField	setText(String text)	void	Sets the text field content.
JTextField	getText()	String	Returns the text field content.
JCheckBox	setSelected(boolean b)	void	Sets the check state.
JCheckBox	isSelected()	boolean	Returns the check state.

1. What is Swing in Java?

Answer: Swing is a part of Java's **Java Foundation Classes (JFC)** that provides a set of lightweight GUI components for building desktop applications. It is platform-independent and more flexible than AWT.

2. What are the key differences between AWT and Swing?

Answer: Swing is lightweight, platform-independent, and provides a richer set of components than AWT. AWT components are heavyweight as they rely on native system components, while Swing components are purely Java-based.

3. What are the main features of Swing?

Answer: Swing provides lightweight components, pluggable look and feel, MVC architecture (not discussing here), event-driven programming, and platform independence.

4. What is a JFrame in Swing?

Answer: JFrame is the top-level container in Swing that represents a window. It provides a title bar, minimize, maximize, and close buttons.

5. What is event handling in Java Swing?

Answer: Event handling in Swing is the process of responding to user interactions like clicks, key presses, and mouse movements using event listeners, event classes, and adapter classes.

6. What are Event Listeners in Swing?

Answer: Event listeners are interfaces that define methods to handle user interactions. Common listeners include ActionListener (for button clicks), MouseListener (for mouse events), and KeyListener (for keyboard events).

7. What are Event Classes in Java Swing?

Answer: Event classes represent specific types of user interactions. Examples include ActionEvent (for button clicks), KeyEvent (for key presses), and MouseEvent (for mouse interactions).

8. What are Adapter Classes in Swing?

Answer: Adapter classes are abstract classes that provide empty implementations of event listener methods, allowing developers to override only necessary methods instead of implementing all methods in an interface.

9. What is Layout Management in Swing?

Answer: Layout management refers to the way components are arranged inside a container. Swing provides layout managers like FlowLayout, BorderLayout, GridLayout, and BoxLayout to control component positioning.

10. What are Basic Swing Components?

Answer: Basic Swing components include JLabel (for displaying text), JButton (for clickable buttons), JTextField (for text input), JTextArea (for multi-line input), and JCheckBox (for checkboxes).

11. What is the difference between JLabel and JTextField?

Answer: JLabel is used for displaying non-editable text, whereas JTextField allows users to enter and edit a single line of text.

12. What is the difference between JPanel and JFrame?

Answer: JFrame is a top-level container that represents a window, while JPanel is a lightweight container used for grouping components inside a JFrame.

13. What is the purpose of JOptionPane in Swing?

Answer: JOptionPane is a utility class used for displaying simple dialog boxes like message dialogs, input dialogs, and confirmation dialogs.

14. What is a JComboBox and how is it different from JList?

Answer: JComboBox is a dropdown list that allows users to select a single item, whereas JList displays multiple items at once and can support multiple selections.

15. How does Swing support graphics and 2D shapes?

Answer: Swing allows drawing 2D shapes using the Graphics and Graphics2D classes. The paintComponent method in JPanel is used for custom drawing.

16. How can colors be applied in Swing components?

Answer: Colors in Swing can be applied using the setForeground method for text color and setBackground for the component background. The Color class provides predefined colors.

17. What are the different types of Swing Containers?

Answer: Swing containers are divided into top-level containers (JFrame, JDialog, JApplet) and intermediate containers (JPanel, JScrollPane).

18. What is the difference between JTextArea and JTextField?

Answer: JTextField allows input of a single line of text, whereas JTextArea supports multiple lines of text input with scrollable features.

19. What is JScrollPane and when is it used?

Answer: JScrollPane is used to add scrolling capabilities to components like JTextArea, JList, or large panels that do not fit within the visible area of the container.

20. What is the importance of SwingUtilities.invokeLater in Swing applications?

Answer: SwingUtilities.invokeLater ensures that GUI updates happen on the Event Dispatch Thread (EDT), preventing concurrency issues in Swing applications.

These questions cover the theoretical aspects of Java Swing GUI programming based on the provided syllabus.

JDBC

JDBC (Java Database Connectivity) is an API in Java used to connect and interact with databases. It provides a standardized method for executing SQL queries and managing database connections.

Characteristics of JDBC

- **API-Based**: Provides a set of classes and interfaces to interact with databases.
- **Platform Independent**: Works across different database systems.
- **Encapsulation**: Hides database-specific details behind a common API.
- **Driver Manager**: Manages different database drivers for seamless connectivity.

JDBC driver types

JDBC uses drivers to establish connections with databases. There are four types:

Type 1: JDBC-ODBC Bridge Driver

- **Uses**: Connects Java applications to databases via ODBC.
- **Pros**: Easy to use, available by default in older JDK versions.
- **Cons**: Dependent on native ODBC drivers, platform-dependent, low performance.

Type 2: Native API Driver

- **Uses**: Uses vendor-specific libraries to connect to databases.
- **Pros**: Faster than Type 1, supports database-specific features.
- **Cons**: Requires native library installation, not portable.

Type 3: Network Protocol Driver

- **Uses**: Uses a middleware server to communicate with the database.
- **Pros**: No need for client-side database libraries, better portability.
- **Cons**: Requires an intermediate server, can add network overhead.

Type 4: Thin Driver (Pure Java Driver)

- **Uses**: Communicates directly with the database using Java.
- **Pros**: Best performance, fully platform-independent, no additional software required.
- **Cons**: Database-specific; requires a separate driver for each database.

Typical uses of JDBC

- Connecting Java applications to relational databases.
- Performing CRUD (Create, Read, Update, Delete) operations.
- Managing transactions (commit/rollback).
- Executing stored procedures and functions.
- Implementing database-driven applications like web applications, desktop applications, and data analytics tools.

JDBC configuration

Steps to Configure JDBC:

1. Load JDBC Driver (For newer versions, automatic loading is supported):

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

2. Establish Connection:

```
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname", "user", "password");
```

3. Create Statement Object:

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

4. Execute SQL Query:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

5. Process Results:

```
while(rs.next()) {
    System.out.println(rs.getString("name"));
}
```

6. Close Connection:

Working with JDBC statements

1. Statement

Used for executing simple SQL queries.

Example:

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

ResultSet rs = stmt.executeQuery("SELECT * FROM
employees");
```

2. PreparedStatement

Used for executing parameterized queries.

More secure (prevents SQL injection) and improves performance.

Example:

```
PreparedStatement pstmt =
con.prepareStatement("SELECT * FROM users WHERE
id = ?");

pstmt.setInt(1, 101);

ResultSet rs = pstmt.executeQuery();
```

3. CallableStatement

Used for executing stored procedures.

Example:

```
CallableStatement cstmt =
con.prepareCall("{call getUser(?)}");

cstmt.setInt(1, 102);

ResultSet rs = cstmt.executeQuery();
```

6. Query Execution

Types of Query Execution Methods:

- **executeQuery():** Used for SELECT queries; returns a ResultSet.
- **executeUpdate():** Used for INSERT, UPDATE, DELETE queries; returns an integer indicating affected rows.
- **execute():** Used for dynamic SQL queries; returns a boolean**.**

Example:

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

ResultSet rs = stmt.executeQuery("SELECT * FROM
products");

while(rs.next()) {

System.out.println(rs.getString("product_name")
);

}
```

7. Scrollable and Updatable Result Sets

By default, a ResultSet is forward-only and read-only. We can make it scrollable and updatable:

Creating Scrollable & Updatable ResultSet:

```
Statement stmt = con.createStatement(

    ResultSet.TYPE_SCROLL_INSENSITIVE,  //
Enables scrolling

    ResultSet.CONCUR_UPDATABLE          //
Enables updates

);

ResultSet rs = stmt.executeQuery("SELECT * FROM
employees");
```

Navigating ResultSet:

```
rs.first();    // Moves to first row

rs.last();     // Moves to last row

rs.previous(); // Moves one row back

rs.absolute(3); // Moves to third row
```

Updating ResultSet:

```
rs.updateString("name", "John Doe");

rs.updateRow();
```

8. Row Sets

A RowSet is an extension of ResultSet that provides additional features like caching and disconnected access. Types include:

Types of RowSets:

- JdbcRowSet: Connected RowSet (like ResultSet but scrollable and updatable).
- CachedRowSet: Disconnected, allows modification even when the database is not connected.
- WebRowSet: Used for XML data.
- FilteredRowSet: Allows filtering rows based on criteria.
- JoinRowSet: Joins data from multiple RowSets.

Example of CachedRowSet:

```
CachedRowSet crs =
RowSetProvider.newFactory().createCachedRowSet(
);

crs.setUrl("jdbc:mysql://localhost:3306/dbname"
);

crs.setUsername("root");

crs.setPassword("password");

crs.setCommand("SELECT * FROM users");

crs.execute();
```

1. Scrollable Result Sets:

A **Scrollable Result Set** allows you to move the cursor in any direction (forward, backward, or arbitrary) through the data. This is a significant advantage over the default **forward-only result set**, which only allows moving forward through the data.

Key Concepts:

- **Type of Scrollable Result Set**:
    - ○ **TYPE_FORWARD_ONLY**: The default type; you can only scroll forward.

- **TYPE_SCROLL_INSENSITIVE**: Allows scrolling in both directions (forward and backward), but the result set is insensitive to changes made by other users.
- **TYPE_SCROLL_SENSITIVE**: Similar to TYPE_SCROLL_INSENSITIVE, but changes made by other users to the data are reflected in the result set.

Scroll Methods:

- next(): Moves the cursor to the next row.
- previous(): Moves the cursor to the previous row.
- first(): Moves the cursor to the first row.
- last(): Moves the cursor to the last row.
- absolute(int row): Moves the cursor to the specified row, counting from the first row.
- relative(int rows): Moves the cursor a specified number of rows relative to the current position.
- beforeFirst(): Moves the cursor before the first row.
- afterLast(): Moves the cursor after the last row.

Advantages of Scrollable Result Sets:

- Flexibility: Navigate the result set freely.
- Allows processing large datasets by jumping to specific rows.

Disadvantages:

- Performance overhead: Consumes more memory and resources.
- Not suitable for small datasets where forward-only navigation is sufficient.

2. Updatable Result Sets:

Updatable Result Sets allow you to update, insert, or delete records from the database directly via the result set, without needing to close the result set and execute a separate SQL command.

Key Concepts:

- **Types of Updatable Result Sets**:
  - **TYPE_FORWARD_ONLY**: Default for forward-only result sets, which can be updatable if the database allows.
  - **TYPE_SCROLL_INSENSITIVE** and **TYPE_SCROLL_SENSITIVE**: Can also be updatable depending on how the result set is created.
- **Methods to Update the Result Set**:
  - updateXxx(int columnIndex, Xxx value): Used to update a specific column.
  - updateXxx(String columnLabel, Xxx value): Similar to the above, but uses column name instead of index.
  - insertRow(): Inserts a new row into the result set.
  - updateRow(): Updates the current row in the result set.
  - deleteRow(): Deletes the current row from the result set.
  - refreshRow(): Refreshes the current row, reflecting changes made outside the result set.

Advantages of Updatable Result Sets:

- Simplifies coding as it allows direct modification of database rows via JDBC.
- Reduces the number of required SQL statements.

Disadvantages:

- Can be resource-intensive.
- Not all databases support updatable result sets.
- The result set's data must be editable (e.g., not a view or read-only table).

3. RowSet:

A **RowSet** is a more advanced concept in JDBC that extends ResultSet and provides additional features like scrollability, updatability, and the ability to be disconnected from the database while still providing access to the data.

Types of RowSets:

- **JdbcRowSet**: A standard implementation of RowSet, which works with JDBC connections and allows working in a disconnected mode.
- **CachedRowSet**: A type of RowSet that works in a disconnected mode and caches rows for offline processing.
- **WebRowSet**: A RowSet for web applications that can convert the data into an XML format.
- **FilteredRowSet**: A RowSet that allows filtering the data returned by the underlying data source.
- **JoinRowSet**: Allows joining data from multiple RowSet objects.

Key Features of RowSets:

- **Disconnected Operation**: RowSets can work without an open connection, making them ideal for client-server applications where the connection is intermittent.
- **Auto-closure of Connection**: Automatically closes the database connection once the data has been fetched.
- **Iterate and Update**: Allows navigating through the data, updating it, and re-connecting to the database if necessary.

Advantages of RowSets:

- **Lightweight**: More efficient than traditional ResultSet.
- **Disconnected**: Allows processing data offline and re-establishing connections when needed.
- **Flexibility**: They can be easily manipulated, and data can be retrieved from a variety of sources (e.g., databases, XML files).

Disadvantages:

- May not always be supported by all databases or environments.
- Not as flexible in highly dynamic scenarios as direct ResultSet usage.

4. Comparison of Result Set vs. RowSet:

- **Connection Dependency**: ResultSet requires an active database connection, while RowSet can be disconnected and work offline.
- **Usage**: ResultSet is suitable for cases where real-time data updates and direct database interaction are needed. RowSet is ideal when offline processing and greater flexibility are required.

CallableStatement in JDBC:

A **CallableStatement** is an extension of the PreparedStatement interface in JDBC. It is used to execute **SQL stored procedures** or **functions** in a database. Callable statements allow for both input and output parameters, making them essential for working with complex database functions.

- **Key Features**:
  - Executes stored procedures or functions in the database.
  - Can handle input and output parameters, such as **IN**, **OUT**, and **INOUT**.
  - Helps improve performance for repeated operations on the database by calling precompiled SQL code.
- **Typical Usage**:
  - Call a stored procedure that may return results or alter the database (e.g., CALL my_procedure(?, ?)).
  - Bind parameters using setXXX() methods and retrieve results with getXXX() methods.

Method	Purpose	Explanation
createStatement()	Create a Statement object for executing SQL queries.	Used for executing static SQL queries that do not require input parameters.
prepareStatement(String sql)	Create a PreparedStatement object for executing precompiled SQL queries.	Prepares a SQL statement that can be executed multiple times with different parameters.
prepareCall(String sql)	Create a CallableStatement object for executing SQL stored procedures.	Used to call stored procedures with input and output parameters.
getMetaData()	Retrieve metadata about the database.	Provides details about the database such as tables, columns, and other objects.
commit()	Commit a transaction.	Saves all changes made during the current transaction to the database.
rollback()	Rollback a transaction.	Reverts all changes made during the current transaction, undoing the commit.
setAutoCommit(boolean autoCommit)	Enable or disable auto-commit for transactions.	Controls whether each statement executes in a transaction (commit after each query) or not.
close()	Close the connection to the database.	Releases database resources and closes the connection when no longer needed.
isReadOnly()	Check if the connection is read-only.	Determines if the database connection is set to allow updates or only reading data.
setReadOnly(boolean readOnly)	Set the connection to be read-only.	Restricts the connection to only read operations and prevents updates or inserts.

1. RMI Definition:

Remote Method Invocation (RMI) is a Java API that allows an object to invoke methods on an object running on another machine in a network. It facilitates communication between applications running on different JVMs (Java Virtual Machines), typically across a network. RMI abstracts the complexities of network programming by allowing remote calls to methods as if they were local.

Roles of Client and Server:

- **Client:** The client initiates a request to invoke a method on the remote object. It accesses the remote object via a stub, which acts as a local representative of the object.
- **Server:** The server creates and registers remote objects with the RMI registry, making them available to clients. The server listens for requests from clients and processes them using the implementation of the remote object.

Remote Method Calls:

The process of remote method calls in RMI involves the following steps:

1. **Client calls a method** on a remote object via a stub.
2. The **stub sends the request** to the remote object on the server.
3. The **server receives the call** and calls the corresponding method on the remote object.
4. The result is returned to the client through the stub.

Stubs and Skeletons:

Stub

- A stub is a client-side proxy that represents the remote object locally. It acts as a gateway for the client to interact with the remote object as if it were a local object.

Role:

- Communication: The stub is responsible for handling all the communication between the client and the server. It ensures that the method calls made by the client are properly forwarded to the remote server where the actual object resides.
- Method Invocation: When a client calls a method on the stub, the stub takes care of packaging (marshalling) the method parameters into a format that can be transmitted over the network.
- Forwarding Call: The stub then sends this packaged data over the network to the server where the remote object resides.
- Receiving Response: After the remote method is executed on the server, the stub receives the result, unpacks (unmarshalls) it, and provides it to the client as if it were a local method call.

Skeleton

- A skeleton is the server-side counterpart of the stub. It is an intermediary that bridges the communication between the stub (client-side) and the actual remote object (server-side).

Role:

- Receiving Call: The skeleton receives the method call from the stub, which includes the method name and the marshalled parameters.
- Unpacking Parameters: It unpacks (unmarshalls) the parameters sent by the stub so that they can be used to invoke the corresponding method on the remote object.
- Invoking Method: The skeleton then invokes the method on the actual remote object with the unpacked parameters.
- Sending Result: After the method is executed, the skeleton packages (marshalls) the result and sends it back over the network to the stub. The stub then unmarshalls this result and provides it to the client.

Example Workflow

Here's a simplified example to illustrate the interaction between the stub and skeleton:

1. **Client Side:**
   - The client calls a method calculateSum(a, b) on the stub.
   - The stub packages (marshalls) the parameters a and b and sends them over the network to the server.
2. **Server Side:**
   - The skeleton on the server receives the method call with the packaged parameters.
   - The skeleton unpacks (unmarshalls) the parameters a and b.
   - The skeleton invokes the method calculateSum(a, b) on the remote object.
   - The remote object executes the method and returns the result sum.
3. **Back to Client:**
   - The skeleton packages (marshalls) the result sum and sends it back to the stub.
   - The stub receives the result, unpacks (unmarshalls) it, and returns it to the client.

Stubs and Parameter Marshalling/Unmarshalling:

Stubs: These are client-side objects that represent remote objects. They allow the client to interact with remote objects as though they are local.

Parameter Marshalling

- Parameter marshalling is the process of converting method arguments (parameters) into a format suitable for transmission over a network.

Role:

- **Preparation:** When a client calls a remote method, the arguments need to be prepared for transmission to the server. This preparation involves converting (serializing) the data into a format that can be sent over a network protocol.

- **Serialization:** The method arguments are serialized into a byte stream, which is a linear representation of the data. This byte stream can travel across the network to the remote server.

- **Transmission:** The serialized byte stream is transmitted from the client to the server over the network.

Unmarshalling

- Unmarshalling is the reverse process of marshalling. It involves converting the serialized byte stream back into the original Java objects at the receiving end (server-side).

Role:

- **Reception:** Once the serialized byte stream reaches the server, it needs to be unpacked (unmarshalled) to retrieve the original method arguments.

- **Deserialization:** The byte stream is deserialized, converting it back into the original Java objects. This allows the server to invoke the corresponding method with the correct arguments.

- **Invocation:** The server invokes the remote method using the unmarshalled parameters, ensuring that the method operates as if it were being called locally.

RMI Programming Model:

The RMI programming model includes:

1. **Remote Interfaces:** These are Java interfaces that define the methods that can be invoked remotely. A remote interface must extend java.rmi.Remote.

2. **Remote Objects:** These are objects whose methods are designed to be invoked remotely. They implement the remote interface and extend java.rmi.server.UnicastRemoteObject for automatic export.

3. **RMI Registry:** A registry where remote objects are registered, allowing clients to look up and retrieve references to remote objects.

RMI Registry:

The **RMI Registry** is a service that stores the names and references of remote objects. It allows clients to look up remote objects using names and retrieve their references. The RMI registry is typically started using the rmiregistry command and runs on a well-known port (1099 by default).

Parameters and Return Values in Remote Methods:

- **Parameters:** When calling remote methods, arguments are sent over the network. These parameters must be serializable (i.e., they must implement java.io.Serializable).

- **Return Values:** Similar to parameters, return values are serialized and sent back to the client. They must also be serializable.

Remote Object Activation:

Parameters:

- **Definition:** When calling remote methods in RMI, arguments (parameters) are transmitted over the network.
- **Serializable:** Parameters must be serializable, which means they must implement the java.io.Serializable interface. This ensures that the objects can be converted into a byte stream for transmission.
- **Process:**
  - **Marshalling:** Parameters are serialized into a byte stream before being sent over the network.
  - **Transmission:** The serialized byte stream is sent from the client to the server.
  - **Unmarshalling:** The server receives the byte stream and deserializes it back into the original objects for use in the remote method.

Return Values:

- **Definition:** Similar to parameters, return values from remote methods are serialized and sent back to the client.
- **Serializable:** Return values must also be serializable to ensure they can be converted into a byte stream and transmitted over the network.
- **Process:**
  - **Marshalling:** The return value is serialized into a byte stream after the remote method execution.
  - **Transmission:** The serialized byte stream is sent back to the client from the server.
  - **Unmarshalling:** The client receives the byte stream and deserializes it back into the original object, providing the result of the remote method call.

Remote Object Activation

Definition:

- Remote object activation allows an object to be created on demand. This means the object does not need to be running when the client first connects.

Activation System:

- **Purpose:** The activation system ensures that a remote object is activated (created and initialized) when needed. This helps manage resources efficiently by only activating objects when they are required.
- **Components:**
  - **ActivationGroup:** This class is used to activate objects. It acts as a container for activated objects and manages their lifecycle.
  - **ActivationMonitor:** This class monitors and manages the activation of remote objects, ensuring they are activated when needed.

. CORBA vs. RMI:

Feature	CORBA	RMI
Architecture	Centralized object request broker.	Java-based, relies on stubs and skeletons.
Protocols	Uses IIOP (Internet Inter-ORB Protocol).	Uses JRMP (Java Remote Method Protocol).
Ease of Use	More complex due to platform independence.	Easier to use for Java-based applications.
Performance	Slower due to additional abstraction layers.	Faster for Java-to-Java communication.
Typical Use Cases	Cross-platform communication, legacy systems.	Primarily used in Java-based environments.

RMI implementation

1. Set Up the Environment

- Ensure Java is installed on both devices (at least Java 8 or higher).
- Ensure both devices can communicate over the network.

2. Create the Remote Interface (Shared between Client and Server)

The remote interface defines the methods that the client can invoke remotely.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

3. Implement the Remote Object (Server Side)

The server implements the remote interface and provides the actual method implementation.

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class HelloImpl extends UnicastRemoteObject implements Hello {

    protected HelloImpl() throws RemoteException {
        super();
    }

    @Override
    public String sayHello() throws RemoteException {
            return "Hello from the Server!";

        }

    }
```

4. Set Up the RMI Server (Server-Side Code)

The server creates an instance of the remote object and binds it to the RMI registry.

```
import java.rmi.Naming;

import java.rmi.registry.LocateRegistry;


public class RMIServer {


    public static void main(String[] args) {

        try {

            // Create the registry on port 1099 (default)

            LocateRegistry.createRegistry(1099);


            // Create the remote object and bind it to the registry

            HelloImpl hello = new HelloImpl();

            Naming.rebind("rmi://<Server_IP>/Hello", hello);

            System.out.println("Server is ready.");

        } catch (Exception e) {

            System.out.println("Server exception: " + e.getMessage());

            e.printStackTrace();

        }

    }

}
```

Explanation:

- LocateRegistry.createRegistry(1099): Creates the RMI registry.
- Naming.rebind("rmi://<Server_IP>/Hello", hello): Binds the remote object to the RMI registry, where <Server_IP> is the server's IP address.

5. Start the RMI Registry on the Server Machine

On the server machine, start the RMI registry in the terminal:

```
rmiregistry 1099
```

6. Create the Client (Client-Side Code)

The client connects to the RMI registry and invokes remote methods.

```
import java.rmi.Naming;


public class RMIClient {


    public static void main(String[] args) {

        try {

            // Connect to the RMI registry on the server's IP address

            Hello hello = (Hello) Naming.lookup("rmi://<Server_IP>/Hello");


            // Call the remote method

            String message = hello.sayHello();

            System.out.println("Message from server: " + message);

        } catch (Exception e) {

            System.out.println("Client exception: " + e.getMessage());

            e.printStackTrace();

        }

    }

}
```

7. Running the Application

- **Step 1 (Server Side):**
  - Compile the server-side code:

    ```
    javac Hello.java HelloImpl.java RMIServer.java
    ```

  - Run the server:

    ```
    java RMIServer
    ```

- **Step 2 (Client Side):**
  - Compile the client-side code:

    ```
    javac RMIClient.java
    ```

  - Run the client:

    ```
    java RMIClient
    ```

8. Ensure Network Communication

- Ensure the firewall or security settings on both devices allow communication on port 1099 (or the custom port).
- The client should be able to connect to the server's IP and port.

Access the database from the RMI application you can use JDBC. The following example is for checking the ticket availability for Airline Reservation System. A ung example captures the flight number on clicking the "seat availability" Button, a method on server is called. This method display the flight number on the server and return the number of seats available for the specified flight. The number is then displayed on the user interface at the Client. If information for the specified flight is not available, the number returned is zero.

Steps to Implement:

1. **Create the Remote Interface** - Define methods for fetching seat availability.
2. **Implement the Remote Server** - Implement the interface, connect to the database, and return seat availability.
3. **Create the Client** - Call the remote method and display the result.
4. **Setup Database** - Create a flights table with flight_number and available_seats.

Remote Interface (AirlineService.java)

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AirlineService extends Remote {
    int checkSeatAvailability(String flightNumber)
throws RemoteException;
}
```

Server Implementation (AirlineServer.java)

```
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;

public class AirlineServer extends UnicastRemoteObject
implements AirlineService {
    private Connection conn;

    public AirlineServer() throws RemoteException {
        super();
        try {
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:33
06/airline_db", "root", "password");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public int checkSeatAvailability(String
flightNumber) throws RemoteException {
        int seats = 0;
        try {
            String query = "SELECT available_seats
FROM flights WHERE flight_number = ?";
            PreparedStatement stmt =
conn.prepareStatement(query);
            stmt.setString(1, flightNumber);
            ResultSet rs = stmt.executeQuery();

            if (rs.next()) {
                seats = rs.getInt("available_seats");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        System.out.println("Checked availability for
Flight: " + flightNumber + " -> Seats Available: " +
seats);
        return seats;
    }

    public static void main(String[] args) {
        try {
            AirlineService service = new
AirlineServer();

Naming.rebind("rmi://localhost/AirlineService",
service);
            System.out.println("Airline Server is
running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client Application (AirlineClient.java)

```
import java.rmi.*;

public class AirlineClient {

    public static void main(String[] args) {

        try {

            AirlineService service = (AirlineService) 
Naming.lookup("rmi://localhost/AirlineService");

            String flightNumber = "AA101"; // Example 
flight number

            int seats = 
service.checkSeatAvailability(flightNumber);

            System.out.println("Seats available for 
Flight " + flightNumber + ": " + seats);

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

Database Setup (MySQL)

```
CREATE DATABASE airline_db;

USE airline_db;


CREATE TABLE flights (

    flight_number VARCHAR(10) PRIMARY KEY,

    available_seats INT

);


INSERT INTO flights VALUES ('AA101', 50), ('BB202', 
30);
```

How to Run:

1. Start **MySQL** and ensure the airline_db database exists.
2. Compile files:

```
javac AirlineService.java AirlineServer.java 
AirlineClient.java
```

3. Start the **RMI registry** in a separate terminal:

```
rmiregistry
```

4. Run the **Server**:

```
java AirlineServer
```

5. Run the **Client**:

```
java AirlineClient
```