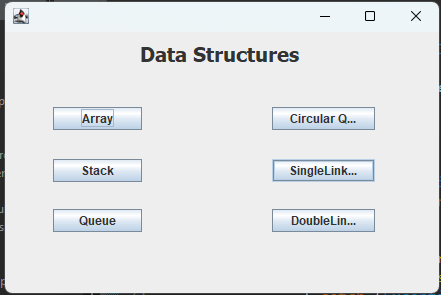
**LinearDSA Java Application**

This Java program creates a simple graphical user interface (GUI) using Swing to manage a collection of common linear data structures, including Queue, Circular Queue, Stack, Linked Lists, and Arrays.



**Key Features:**

1. **Frame Creation**:
   * The LinearDSA class extends JFrame, which is the main window of the application.
   * The frame's layout is set to null, allowing for manual placement of components like buttons and labels.
2. **Buttons**:
   * There are six buttons, each representing a different data structure: Queue, Circular Queue, Stack, Double Linked List, Single Linked List, and Array.
   * Each button is associated with an ActionListener, which defines the action to be taken when the button is clicked.
   * When a button is clicked, it calls the close() method to close the current window and then opens a new window related to the selected data structure.
3. **Closing the Window**:
   * The close() method simulates closing the current window. This is achieved using WindowEvent and Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent().
4. **Content Panel**:
   * The content panel (contentPane) holds all the components, and its layout is managed manually using setLayout(null), which means no automatic layout manager is used. The components' positions are manually set using setBounds(x, y, width, height).
5. **Label**:
   * A JLabel titled "Data Structures" is added at the top with a bold font to give the GUI a title.

**Example Flow:**

* When the application starts, a window with a title "Data Structures" appears, showing buttons for the six data structures.
* When a user clicks a button (e.g., "Queue"), the current window closes and the program opens a new window related to the Queue data structure.

**Recommendations for Improvement:**

1. **Modularize the Data Structure Classes**: Instead of directly creating Array objects for each data structure type, it's better to create specific classes for each data structure (e.g., QueueFrame, StackFrame), which would then handle the visualization and functionality of that data structure.
2. **Improve Layout Management**: Instead of using setLayout(null), which can be error-prone, consider using layout managers like GridLayout, FlowLayout, or BorderLayout for better automatic placement and resizing of components.
3. **Window Resizing**: The fixed window size (450x300) could cause issues with window resizing or varying screen sizes. You might want to make the window resizable or adjust components dynamically based on window size.
4. **Close Operation Enhancement**: The close() method could be enhanced by using setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE) if the program should terminate when the window is closed.

**Key Components of the Program:**

1. **Main Frame (LinearDSA):**
   * The main window of the application extends JFrame and provides the interface for users to interact with.
   * The frame contains a JPanel which holds all UI components like buttons and labels. This panel uses a null layout, meaning components are positioned manually using setBounds(x, y, width, height).
2. **Buttons:**
   * The GUI includes six buttons, each representing a different linear data structure:
     + **Queue**
     + **Circular Queue**
     + **Stack**
     + **Double Linked List**
     + **Single Linked List**
     + **Array**
   * Each button is associated with an ActionListener that triggers the opening of a new window corresponding to the selected data structure. For example, clicking the "Queue" button opens a new window for interacting with a queue.
3. **Window Closing Mechanism:**
   * The application includes a close() method that closes the current window when a button is pressed. This method uses WindowEvent and Toolkit to post a closing event to the system event queue.
4. **Title Label:**
   * A JLabel titled "Data Structures" is added at the top of the frame. It uses a bold font to make the label prominent and acts as a title for the window.
5. **Data Structure Classes:**
   * The Array class (and other data structure classes like Stack, Queue, etc.) are assumed to exist and provide the core functionality for managing the respective data structures. These classes are instantiated with specific parameters (like "Queue" for a queue) when a user selects a data structure from the main window.

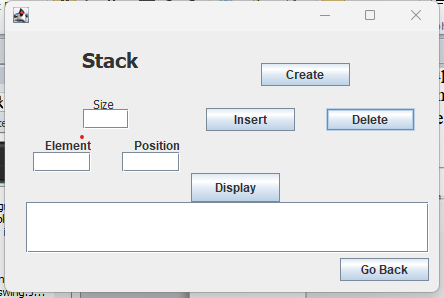
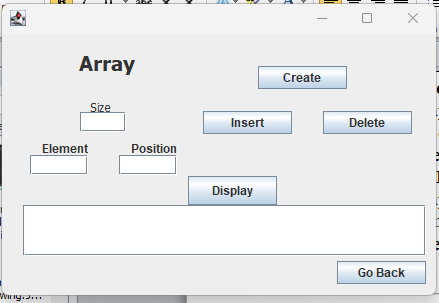
**Application Flow:**

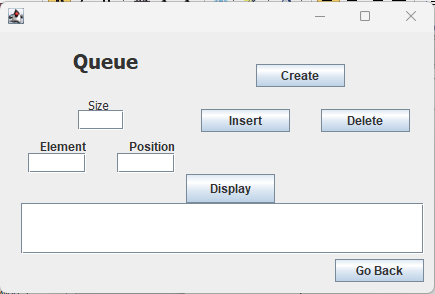
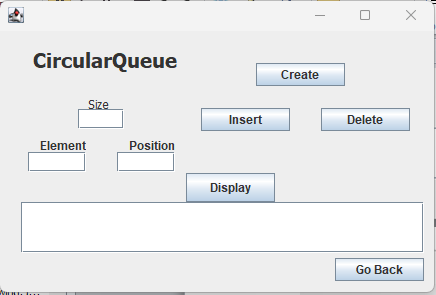
1. **Launch Application:**
   * The application is launched by invoking the LinearDSA class in the main() method.
   * The main window appears with the title "Data Structures" and six buttons.
2. **User Interaction:**
   * Users click a button corresponding to the desired data structure (e.g., "Queue").
   * The program closes the current window and opens a new window related to the selected data structure. This new window is expected to allow further operations on the chosen data structure (e.g., adding/removing elements from a queue).
3. **Window Closing:**
   * The close() method ensures that the main window closes before transitioning to the new data structure window. It uses a WindowEvent to trigger the closing of the window.

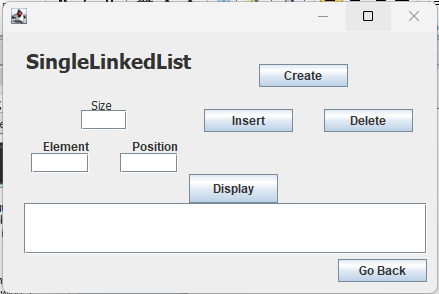
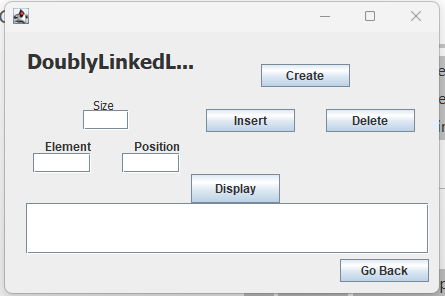
**Example: Queue Button Interaction**

When the user clicks the **Queue** button:

* The close() method is called to close the current window.
* A new Array object is created with the string "Queue" as an argument (new Array("Queue")).
* The Array window is then made visible, showing the interface for interacting with a queue (the specifics of which are assumed to be implemented within the Array class).

****

**** ****

**** ****

**Overview:**

The LinearDSA Java application is a graphical user interface (GUI) built using the Swing framework. It provides a user-friendly interface to interact with and visualize several linear data structures, including arrays, stacks, queues, circular queues, and linked lists (both singly and doubly linked). The application allows users to select a data structure, after which a new window corresponding to the selected data structure opens for further interaction.

The DSA Java application provides a user interface for managing linear data structures such as arrays or lists. Using Java Swing, this GUI allows users to create, insert, delete, and display elements within a specific data structure. The program dynamically updates the data structure based on user input and provides feedback through dialog boxes.

**Key Components of the Program:**

1. **Frame (DSA):**
   * This class creates the main window (frame) using JFrame where users interact with different operations on a linear data structure (e.g., array).
   * The window includes buttons, text fields, and labels for user interaction.
2. **User Interface (UI) Components:**
   * **Text Fields:**
     + txt1: Input field for the size of the data structure.
     + txt2: Input field for the element to be inserted or deleted.
     + txt3: Input field for the position at which the element should be inserted or deleted.
     + txt4: Output field that displays the current elements of the data structure after any operation (display).
   * **Buttons:**
     + **Go Back**: Returns to the previous main frame (LinearDSA).
     + **Create**: Initializes the data structure with a specified size.
     + **Insert**: Inserts an element at a specific position in the data structure.
     + **Delete**: Deletes an element from a specified position.
     + **Display**: Shows the current contents of the data structure in txt4.
3. **Action Listeners:**
   * Each button has an ActionListener attached to it, performing actions like creating a data structure, inserting/deleting elements, or displaying the current state of the structure.
4. **Data Structure Operations:**
   * **Create:** The create method initializes the data structure with the given size.
   * **Insert:** Inserts a value at a given position in the data structure, using 1-based indexing (position - 1).
   * **Delete:** Deletes an element from a specified position in the data structure.
   * **Display:** Displays the current elements in the data structure in txt4.
5. **Dialog Boxes:**
   * **JOptionPane:** Used for showing success or error messages, such as when an element is inserted, deleted, or when the data structure is created.

**Flow of Operations:**

1. **Launch Application:**
   * The main() method initializes and displays the DSA frame.
2. **Creating a Data Structure:**
   * The user enters the desired size of the data structure in txt1 and clicks the **Create** button.
   * The create method initializes the data structure, and a success message is shown.
3. **Inserting Elements:**
   * The user provides a value and a position (using txt2 and txt3), then clicks the **Insert** button.
   * The insert method places the value at the specified position and a confirmation message appears.
4. **Deleting Elements:**
   * The user provides the position of the element to be deleted in txt3 and clicks the **Delete** button.
   * The delete method removes the element from the given position and displays a confirmation message.
5. **Displaying the Data Structure:**
   * The user clicks the **Display** button to view the current elements in the data structure, which are displayed in txt4.

**Key Methods:**

* **create(int size):** Initializes the data structure with a given size.
* **insert(String type, int value, int position):** Inserts an element at the specified position.
* **delete(String type, int position):** Deletes the element at the specified position.
* **display(String type):** Returns a string representing the current state of the data structure.

**Additional Features:**

* **Back Button:** The "Go Back" button closes the current window and opens the main menu (the LinearDSA frame).

**Conclusion:**

The LinearDSA Jframe application provides a simple interface for users to explore different linear data structures. While the current implementation serves as a basic starting point, it could be significantly improved with enhanced modularity, better layout management, and the addition of functionality for each data structure.

The DSA Jframe is a user-friendly interface that allows users to manage linear data structures through a simple set of operations like creating, inserting, deleting, and displaying elements. While functional, there are opportunities for enhancement, such as improving error handling and offering more dynamic data structure management.

**Code:**

**LinearDSA Frame Code:**

import java.awt.EventQueue;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.border.EmptyBorder;

import javax.swing.JLabel;

import javax.swing.JButton;

import javax.swing.JToggleButton;

import javax.swing.JTextField;

import javax.swing.SwingConstants;

import java.awt.event.ActionListener;

import java.awt.event.WindowEvent;

import java.awt.event.ActionEvent;

import java.awt.Font;

import java.awt.Toolkit;

import java.beans.PropertyChangeListener;

import java.beans.PropertyChangeEvent;

public class LinearDSA extends JFrame {

private static final long serialVersionUID = 1L;

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

public void run() {

try {

LinearDSA frame = new LinearDSA();

frame.setVisible(true);

} catch (Exception e) {

e.printStackTrace();

}

}

});

}

public void close() {

WindowEvent closeWindow = new WindowEvent(this, WindowEvent.WINDOW\_CLOSING);

Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(closeWindow);

}

public LinearDSA() {

setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

setBounds(100, 100, 450, 300);

JPanel contentPane = new JPanel();

contentPane.addPropertyChangeListener(new PropertyChangeListener() {

public void propertyChange(PropertyChangeEvent evt) {

}

});

contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));

setContentPane(contentPane);

contentPane.setLayout(null);

JButton btnNewButton = new JButton("Queue");

btnNewButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

DSA array = new DSA("Queue");

array.setVisible(true);

}

});

btnNewButton.setBounds(48, 177, 89, 23);

contentPane.add(btnNewButton);

JButton btnNewButton\_1 = new JButton("Circular Queue");

btnNewButton\_1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

DSA array = new DSA("CircularQueue");

array.setVisible(true);

}

});

btnNewButton\_1.setBounds(267, 75, 103, 23);

contentPane.add(btnNewButton\_1);

JButton btnNewButton\_2 = new JButton("Stack");

btnNewButton\_2.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

DSA array = new DSA("Stack");

array.setVisible(true);

}

});

btnNewButton\_2.setBounds(48, 127, 89, 23);

contentPane.add(btnNewButton\_2);

JButton btnNewButton\_3 = new JButton("DoubleLinkedList");

btnNewButton\_3.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

DSA array = new DSA("DoublyLinkedList");

array.setVisible(true);

}

});

btnNewButton\_3.setBounds(267, 177, 103, 23);

contentPane.add(btnNewButton\_3);

JButton btnNewButton\_4 = new JButton("Array");

btnNewButton\_4.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

DSA array = new DSA("Array");

array.setVisible(true);

}

});

btnNewButton\_4.setBounds(48, 75, 89, 23);

contentPane.add(btnNewButton\_4);

JButton btnNewButton\_5 = new JButton("SingleLinkedList");

btnNewButton\_5.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

DSA array = new DSA("SingleLinkedList");

array.setVisible(true);

}

});

btnNewButton\_5.setBounds(267, 127, 103, 23);

contentPane.add(btnNewButton\_5);

JLabel lblNewLabel = new JLabel("Data Structures");

lblNewLabel.setFont(new Font("Tahoma", Font.BOLD, 20));

lblNewLabel.setBounds(135, 11, 166, 23);

contentPane.add(lblNewLabel);

}

}

**DSA Frame Code:**

import java.awt.EventQueue;

import java.awt.Toolkit;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.border.EmptyBorder;

import javax.swing.JButton;

import java.awt.event.ActionListener;

import java.awt.event.WindowEvent;

import java.awt.event.ActionEvent;

import javax.swing.JLabel;

import javax.swing.JOptionPane;

import javax.swing.JTextField;

import java.awt.Font;

import javax.swing.SwingConstants;

import com.miniproject.linearDS.copy2.LinearDS;

public class DSA extends JFrame {

private static final long serialVersionUID = 1L;

private JPanel contentPane;

private JTextField txt1;

private JTextField txt2;

private JTextField txt3;

private JTextField txt4;

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

public void run() {

try {

DSA frame = new DSA("");

frame.setVisible(true);

} catch (Exception e) {

e.printStackTrace();

}

}

});

}

public void close() {

WindowEvent closeWindow = new WindowEvent(this, WindowEvent.WINDOW\_CLOSING);

Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(closeWindow);

}

/\*\*

\* Create the frame.

\*/

public DSA(String s) {

setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

setBounds(100, 100, 450, 300);

contentPane = new JPanel();

contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));

setContentPane(contentPane);

contentPane.setLayout(null);

LinearDS structures = new LinearDS();

JButton btnNewButton = new JButton("Go Back");

btnNewButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

close();

LinearDSA mainFrame = new LinearDSA();

mainFrame.setVisible(true);

}

});

btnNewButton.setBounds(335, 227, 89, 23);

contentPane.add(btnNewButton);

JButton btnNewButton\_1 = new JButton("Display");

btnNewButton\_1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

txt4.setText(structures.display(s));

}

});

btnNewButton\_1.setBounds(186, 142, 89, 29);

contentPane.add(btnNewButton\_1);

JLabel lblNewLabel = new JLabel("Size");

lblNewLabel.setFont(new Font("Tahoma", Font.PLAIN, 12));

lblNewLabel.setBounds(88, 66, 36, 14);

contentPane.add(lblNewLabel);

txt1 = new JTextField();

txt1.setBounds(78, 78, 46, 20);

contentPane.add(txt1);

txt1.setColumns(10);

JButton btnNewButton\_2 = new JButton("Create");

btnNewButton\_2.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

int num1 = Integer.parseInt(txt1.getText());

structures.create(num1);

JOptionPane.showMessageDialog(null, s + "Created");

}

});

btnNewButton\_2.setBounds(256, 32, 89, 23);

contentPane.add(btnNewButton\_2);

txt2 = new JTextField();

txt2.setBounds(28, 121, 58, 20);

contentPane.add(txt2);

txt2.setColumns(10);

txt3 = new JTextField();

txt3.setBounds(117, 121, 58, 20);

contentPane.add(txt3);

txt3.setColumns(10);

JButton btnNewButton\_3 = new JButton("Insert");

btnNewButton\_3.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

int value = Integer.parseInt(txt2.getText());

int position = Integer.parseInt(txt3.getText());

structures.insert(s, value, position-1);

JOptionPane.showMessageDialog(null,"In "+ s + " Inserted Value:" + value + " at postion:" + position);

}

});

btnNewButton\_3.setBounds(201, 77, 89, 23);

contentPane.add(btnNewButton\_3);

JButton btnNewButton\_4 = new JButton("Delete");

btnNewButton\_4.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

int position = Integer.parseInt(txt3.getText());

structures.delete(s, position-1);

JOptionPane.showMessageDialog(null,"In "+ s + " Deleted at postion:" + position);

}

});

btnNewButton\_4.setBounds(321, 77, 89, 23);

contentPane.add(btnNewButton\_4);

txt4 = new JTextField();

txt4.setBounds(21, 171, 403, 51);

contentPane.add(txt4);

txt4.setColumns(10);

JLabel lblNewLabel\_1 = new JLabel("Element");

lblNewLabel\_1.setBounds(40, 107, 46, 14);

contentPane.add(lblNewLabel\_1);

JLabel lblNewLabel\_3 = new JLabel("Position");

lblNewLabel\_3.setBounds(129, 107, 46, 14);

contentPane.add(lblNewLabel\_3);

JLabel lblNewLabel\_2 = new JLabel(s);

lblNewLabel\_2.setFont(new Font("Tahoma", Font.BOLD, 20));

lblNewLabel\_2.setHorizontalAlignment(SwingConstants.CENTER);

lblNewLabel\_2.setBounds(21, 11, 169, 36);

contentPane.add(lblNewLabel\_2);

}

}

**Implementation of Linear Data Structures:**

package com.miniproject.linearDS.copy2;

import java.util.\*;

public class LinearDS {

Stack<Integer> stack = new Stack<>();

Queue<Integer> queue = new LinkedList<>();

CircularQueue<Integer> cQueue = new CircularQueue<>();

LinkedList<Integer> lList = new LinkedList<>();

LinkedList<Integer> sList = new LinkedList<>();

public

static int[] *a*;

public void create(int size) {

*a* = new int[size];

}

public void insert(String s, int value, int position) {

*a*[position] = value;

switch (s) {

case "Stack":

stack.push(*a*[position]);

break;

case "Queue":

queue.add(*a*[position]);

break;

case "CircularQueue":

cQueue.add(value, position);

break;

case "SingleLinkedList":

sList.add(value);

break;

case "DoublyLinkedList":

lList.add(value);;

break;

case "Array":

*a*[position] = value;

break;

default:

System.***out***.println("Invalid input");

break;

}

}

public void delete(String s, int position) {

switch (s) {

case "Stack":

stack.remove(position);

break;

case "Queue":

queue.remove(position);

break;

case "CircularQueue":

cQueue.remove(position);

break;

case "SingleLinkedList":

sList.remove(position);

break;

case "DoublyLinkedList":

lList.remove(position);

break;

case "Array":

*a*[position] = 0;

break;

default:

System.***out***.println("Invalid input");

break;

}

}

public String display(String s) {

String dis = "";

switch (s) {

case "Stack":

for(int p : stack)

dis += p + " ";

return dis;

case "Queue":

for(int p : queue)

dis += p + " ";

return dis;

case "CircularQueue":

return cQueue.display();

case "SingleLinkedList":

for(int p : sList)

dis += p + " ";

return dis;

case "DoublyLinkedList":

for(int p : lList)

dis += p + " ";

return dis;

case "Array":

for(int p : *a*)

dis += p + " ";

return dis;

default:

return "Invalid input";

}

}

}

**Circular Queue Implementation:**

package com.miniproject.linearDS.copy2;

class Node {

int data;

Node next;

// Constructor to create a new node

public Node(int data) {

this.data = data;

this.next = null;

}

}

public class CircularQueue<Integer> {

Node front, rear;

// Constructor to initialize the queue

public CircularQueue() {

front = rear = null;

}

// Check if the queue is empty

public boolean isEmpty() {

return front == null;

}

// Enqueue operation at specific position

public void add(int data, int position) {

Node newNode = new Node(data);

// If the queue is empty, set the new node as both front and rear

if (isEmpty()) {

front = rear = newNode;

rear.next = front; // Circular link

} else if (position == 0) {

// Insert at the front of the queue (before front)

newNode.next = front;

rear.next = newNode;

front = newNode;

} else {

Node current = front;

int count = 0;

// Traverse the queue to the position

while (current != rear && count < position - 1) {

current = current.next;

count++;

}

// Insert the node at the specific position

newNode.next = current.next;

current.next = newNode;

// If inserted at the rear

if (current == rear) {

rear = newNode;

}

}

}

// Dequeue operation from a specific position

public int remove(int position) {

if (isEmpty()) {

System.***out***.println("Queue is empty!");

return -1;

}

if (position == 0) {

// Dequeue the front

return removeFirst();

}

Node current = front;

int count = 0;

// Traverse the queue to the position

while (current != rear && count < position - 1) {

current = current.next;

count++;

}

// If current is rear, we are removing the rear node

if (current == rear) {

return removeLast();

}

// Remove the node at the position

Node temp = current.next;

current.next = temp.next;

return temp.data;

}

// Dequeue operation from the front (helper method)

private int removeFirst() {

if (front == rear) {

int data = front.data;

front = rear = null;

return data;

} else {

int data = front.data;

front = front.next;

rear.next = front; // Maintain circular link

return data;

}

}

// Dequeue operation from the rear (helper method)

private int removeLast() {

if (front == rear) {

int data = rear.data;

front = rear = null;

return data;

}

Node current = front;

while (current.next != rear) {

current = current.next;

}

int data = rear.data;

rear = current;

rear.next = front; // Maintain circular link

return data;

}

// Display the queue elements

public String display() {

String s = "";

if (isEmpty()) {

System.***out***.println("Queue is empty!");

return "";

}

Node current = front;

while (current != rear) {

s += current.data + " ";

current = current.next;

}

s += current.data;

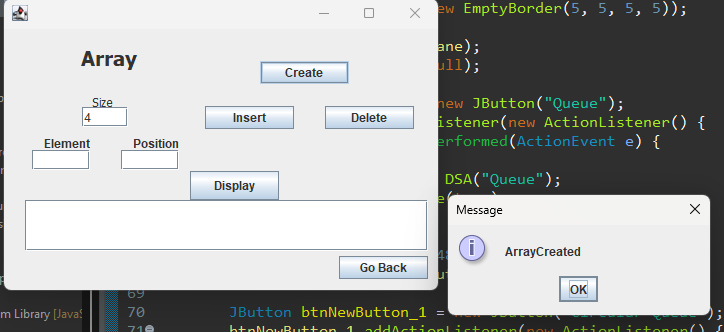
return s;

}

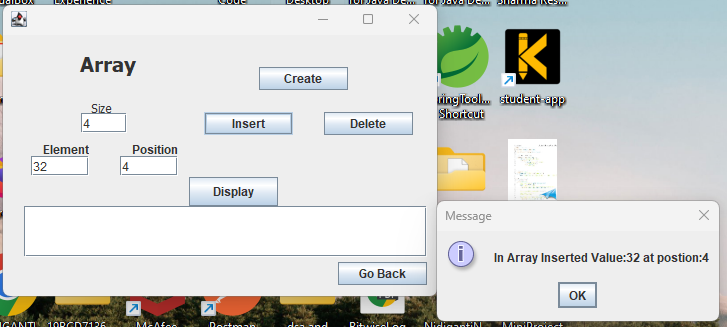
}

**Output:**

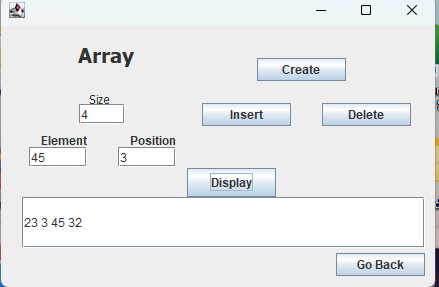
Creating Array:

****

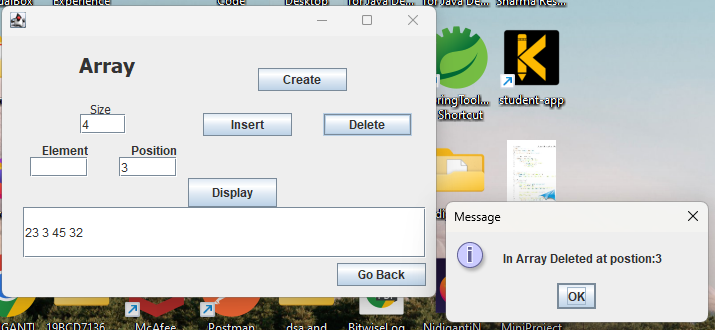
Insertion of values at specific position:

****

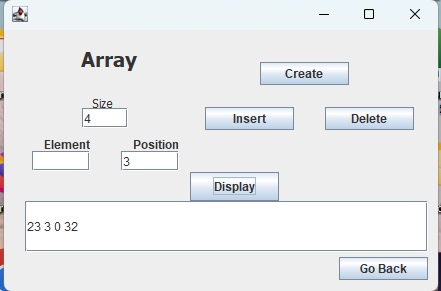
Displaying the elements before Deletion:

****

Deletion based on the Index value:

****

Displaying of Elements after deletion:

****