JAVA Mid 2 QnA

1. a) What are the differences between an Applet and stand alone application? b) What is Swing in Java? How it differs from Applet.

Differences between an Applet and Standalone Application

Feature	Applet	Standalone Application
Execution Environment	Runs in a web browser	Runs directly on the operating system
Deployment	Requires an HTML page and a web browser	Requires an installation on the local machine
Security Restrictions	Runs in a sandbox with restricted permissions	Can have full access to system resources
User Interface	Built using AWT/Swing for GUI	Built using AWT/Swing for GUI
Startup	Initiated by loading the HTML page in a browser	Initiated by running the executable file
Use Cases	Typically used for small web-based applications	Used for a wide range of applications, from small to large
Lifecycle Methods	Has lifecycle methods like `init()`, `start()`, `stop()`, `destroy()`	Does not have predefined lifecycle methods, typically has a `main()` method
Example	Online games, interactive web tools	Desktop applications like text editors, media players

Swing in Java

Swing is a part of Java's standard library that provides a set of lightweight (all-Java language) components for building graphical user interfaces (GUIs). It is built on top of the Abstract Window Toolkit (AWT) but provides more sophisticated and flexible components.

Differences between Swing and Applet

Feature	Swing	Applet
Purpose	Library for creating standalone GUI applications	Special type of program designed to be embedded in web pages

Feature	Swing	Applet
Component Set	Rich set of components like buttons, tables, trees, etc.	Uses AWT components but can also use Swing components for richer interfaces
Look and Feel	Pluggable look-and-feel, can mimic different OS styles	Limited to the look-and-feel provided by the AWT and Swing
Flexibility	Highly flexible and customizable	Limited in terms of deployment and interaction
Usage	Used in standalone applications and applets	Specifically designed for web-based usage within a browser
Architecture	Lightweight components that don't rely on native code	Applets use AWT components by default, which are heavyweight and rely on native code
Event Handling	Event delegation model	Uses the event delegation model, similar to AWT
Example	Desktop applications like IDEs, media players	Small web-based applications like games, calculators

- **Swing** provides a richer set of GUI components compared to AWT, including features like tabbed panes, sliders, tooltips, and more.
- **Swing** components are lightweight as they are written entirely in Java and do not rely on the underlying operating system's windowing system, making them more portable and consistent across different platforms.
- **Applets** are less commonly used today due to security concerns and the advent of more modern web technologies like HTML5, CSS, and JavaScript.

2. With syntax,explain the following utility classes a) String Tokenizer b)Date and Calendar c)Scanner

a) StringTokenizer

The `StringTokenizer` class in Java is used to break a string into tokens. It's part of the `java.util` package. This class is a legacy class retained for compatibility reasons and its use is discouraged in new code. Instead, you should use the `String.split()` method or the `Scanner` class.

Syntax and Example

```java
import java.util.StringTokenizer;

public class StringTokenizerExample {
    public static void main(String[] args) {
        String str = "Hello, how are you?";
        StringTokenizer st = new StringTokenizer(str, " ");

        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

In this example, the string `"Hello, how are you?"` is tokenized using a space (`" "`) as the delimiter.

b) Date and Calendar

The `Date` class represents a specific instant in time, with millisecond precision. However, this class has many deprecated methods and is not preferred for date manipulation. Instead, the `Calendar` class is often used.

Date Syntax and Example

```java
import java.util.Date;

public class DateExample {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(date);
    }
}
```

This example creates a `Date` object representing the current date and time.

Calendar Syntax and Example

```java
import java.util.Calendar;

public class CalendarExample {
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
```

```
        System.out.println("Current Date and Time: " + calendar.getTime());

        // Get individual components
        int year = calendar.get(Calendar.YEAR);
        int month = calendar.get(Calendar.MONTH); // Note: 0-based
        int day = calendar.get(Calendar.DAY_OF_MONTH);
        int hour = calendar.get(Calendar.HOUR_OF_DAY);
        int minute = calendar.get(Calendar.MINUTE);
        int second = calendar.get(Calendar.SECOND);

        System.out.println("Year: " + year);
        System.out.println("Month: " + (month + 1)); // Adding 1 as
Calendar.MONTH is 0-based
        System.out.println("Day: " + day);
        System.out.println("Hour: " + hour);
        System.out.println("Minute: " + minute);
        System.out.println("Second: " + second);
    }
}
```

This example uses the `Calendar` class to get the current date and time, and then extracts individual date/time components.

c) Scanner

The `Scanner` class is used to parse primitive types and strings using regular expressions. It is part of the `java.util` package and is commonly used for reading input from various sources, including the console, files, and strings.

Syntax and Example

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your name:");
        String name = scanner.nextLine();

        System.out.println("Enter your age:");
        int age = scanner.nextInt();

        System.out.println("Enter your height in cm:");
        double height = scanner.nextDouble();
```

```
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height + " cm");

        scanner.close();
    }
}
```

This example demonstrates how to use the `Scanner` class to read different types of input from the console. It reads a string, an integer, and a double, and then prints the values.

- **StringTokenizer**: A legacy class for splitting strings into tokens.
- **Date**: Represents a specific instant in time.
- **Calendar**: Provides methods for date and time manipulation.
- **Scanner**: Used for parsing primitive types and strings from various input sources.

3. a) Differentiate between multiprocessing and multithreading b) What is synchronization and why it is important?

a) Differences between Multiprocessing and Multithreading

Feature	Multiprocessing	Multithreading
Definition	Multiple processes running simultaneously	Multiple threads running within a single process
Memory Usage	Each process has its own memory space	Threads share the same memory space within a process
Communication	Inter-process communication (IPC) is needed, which can be complex	Threads communicate directly through shared memory
Concurrency	Achieves true parallelism as each process can run on a different CPU core	Provides concurrency but not necessarily parallelism (depends on CPU cores)
Isolation	Processes are isolated from each other, reducing risk of data corruption	Threads are not isolated and can interfere with each other if not managed properly
Context Switching	More overhead due to switching between processes	Less overhead as threads are lighter than processes
Creation Time	More time to create a new process	Less time to create a new thread
Use Cases	CPU-intensive tasks, tasks that require isolation	I/O-bound tasks, tasks that require shared data

Feature	Multiprocessing	Multithreading
Example	Running multiple independent programs	Running multiple tasks within a single program (e.g., a web server handling multiple requests)

b) What is Synchronization and Why is it Important?

Synchronization is a mechanism that ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a **critical section**. It is used to prevent race conditions and ensure data consistency when multiple threads or processes access shared resources.

Importance of Synchronization:

1. **Preventing Race Conditions**: When multiple threads or processes access and modify shared data simultaneously, race conditions can occur, leading to inconsistent or incorrect data. Synchronization ensures that only one thread or process can access the critical section at a time, preventing race conditions.
2. **Data Consistency**: Synchronization helps maintain data integrity and consistency by ensuring that shared data is accessed and modified in a controlled manner.
3. **Coordination Between Threads/Processes**: Synchronization is crucial for coordinating the actions of multiple threads or processes. It ensures that the tasks are executed in the correct order and that shared resources are used efficiently and without conflict.
4. **Deadlock Prevention**: Proper use of synchronization techniques can help prevent deadlocks, where two or more threads are waiting for each other to release resources, causing the system to halt.

Synchronization Mechanisms in Java:

1. **Synchronized Methods**: Only one thread can execute a synchronized method of an object at any given time.

```
public synchronized void synchronizedMethod() {
    // Critical section
}
```

2. **Synchronized Blocks**: A synchronized block is more fine-grained and can be used to synchronize a specific part of a method.

```
public void method() {
    synchronized (this) {
        // Critical section
    }
}
```

3. **Locks (java.util.concurrent.locks.Lock)**: Provides more extensive locking operations than `synchronized` blocks/methods.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Example {
    private final Lock lock = new ReentrantLock();

    public void method() {
        lock.lock();
        try {
            // Critical section
        } finally {
            lock.unlock();
        }
    }
}
```

4. **Volatile Keyword**: Used to mark a variable as being stored in main memory. Every read of a volatile variable will be read from the main memory, and every write to a volatile variable will be written to the main memory.

```
private volatile boolean flag;
```

4. a) Explain various layout managers in JAVA. b) Write a program to create a frame window that responds to mouse clicks?

a) Various Layout Managers in Java

Layout managers in Java are used to control the positioning and sizing of components in a container. Here are some of the most commonly used layout managers:

1. **FlowLayout**

- **Description**: Arranges components in a left-to-right flow, much like lines of text in a paragraph.
- **Use Case**: Simple, default layout for small number of components.
- **Example**:

```
setLayout(new FlowLayout());
```

2. **BorderLayout**
   - **Description**: Divides the container into five regions: North, South, East, West, and Center. Each region can contain only one component.
   - **Use Case**: Useful for creating a standard window layout with a header, footer, sidebars, and central content.
   - **Example**:

```
setLayout(new BorderLayout());
add(new Button("North"), BorderLayout.NORTH);
add(new Button("South"), BorderLayout.SOUTH);
add(new Button("East"), BorderLayout.EAST);
add(new Button("West"), BorderLayout.WEST);
add(new Button("Center"), BorderLayout.CENTER);
```

3. **GridLayout**
   - **Description**: Arranges components in a grid with equal-sized cells.
   - **Use Case**: Useful for creating uniform grid layouts like a calculator or button panel.
   - **Example**:

```
setLayout(new GridLayout(3, 2));
```

4. **GridBagLayout**
   - **Description**: A flexible grid-based layout that allows components to span multiple cells and have different sizes.
   - **Use Case**: Useful for creating complex, custom layouts.
   - **Example**:

```
setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 0;
add(new Button("Button 1"), gbc);
```

5. **CardLayout**

- **Description**: Treats each component as a card. Only one card is visible at a time.
- **Use Case**: Useful for implementing tabbed panes, wizard dialogs, etc.
- **Example**:

```
setLayout(new CardLayout());
add(new Button("Card 1"), "Card 1");
add(new Button("Card 2"), "Card 2");
```

6. **BoxLayout**
   - **Description**: Arranges components either vertically (BoxLayout.Y_AXIS) or horizontally (BoxLayout.X_AXIS).
   - **Use Case**: Useful for creating vertical or horizontal stack of components.
   - **Example**:

```
setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
```

b) Program to Create a Frame Window that Responds to Mouse Events

Here is a simple Java program using Swing to create a frame window that responds to mouse events:

```
import javax.swing.*;
import java.awt.event.*;

public class MouseEventExample extends JFrame implements MouseListener {

    public MouseEventExample() {
        setTitle("Mouse Event Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        // Add mouse listener
        addMouseListener(this);

        setVisible(true);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse Clicked at (" + e.getX() + ", " + e.getY()
+ ")");
```

```
    }

    @Override
    public void mousePressed(MouseEvent e) {
        System.out.println("Mouse Pressed at (" + e.getX() + ", " + e.getY()
+ ")");
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        System.out.println("Mouse Released at (" + e.getX() + ", " +
e.getY() + ")");
    }

    @Override
    public void mouseEntered(MouseEvent e) {
        System.out.println("Mouse Entered the frame");
    }

    @Override
    public void mouseExited(MouseEvent e) {
        System.out.println("Mouse Exited the frame");
    }

    public static void main(String[] args) {
        new MouseEventExample();
    }
}
```

Explanation of the Program

- **Imports**: `javax.swing.*` for the Swing components, `java.awt.event.*` for the event handling.
- **Class**: `MouseEventExample` extends `JFrame` and implements `MouseListener`.
- **Constructor**: Sets up the frame properties (title, size, close operation, and location). It also adds the mouse listener to the frame.
- **Mouse Listener Methods**:
  - `mouseClicked()`: Triggered when the mouse is clicked.
  - `mousePressed()`: Triggered when the mouse button is pressed.
  - `mouseReleased()`: Triggered when the mouse button is released.
  - `mouseEntered()`: Triggered when the mouse enters the frame area.
  - `mouseExited()`: Triggered when the mouse exits the frame area.
- **Main Method**: Creates an instance of the `MouseEventExample` class to display the frame.

5. a) What are the limitations of AWT? b) Write a program to create a frame window that responds to key strokes.

a) Limitations of AWT (Abstract Window Toolkit)

1. **Platform Dependence**: AWT components are heavy-weight, meaning they rely on the underlying platform's native GUI components. This can lead to inconsistencies in appearance and behavior across different platforms.
2. **Limited GUI Components**: AWT provides a relatively small set of GUI components compared to more modern frameworks like Swing or JavaFX.
3. **Less Flexibility**: AWT does not support features like pluggable look-and-feel or advanced GUI components (e.g., tables, trees, or text components with rich formatting).
4. **Event Handling Model**: The event handling model in AWT is more cumbersome and less powerful compared to the more flexible and powerful event delegation model introduced in Swing.
5. **Performance**: Due to its reliance on native peer components, AWT can suffer from performance issues, especially when creating complex UIs or handling a large number of components.
6. **Threading Issues**: AWT is not thread-safe, and updating the GUI from outside the Event Dispatch Thread (EDT) can lead to unpredictable behavior and bugs.

b) Program to Create a Frame Window that Responds to Key Events

Here's a simple Java program using Swing to create a frame window that responds to key events:

```
import javax.swing.*;
import java.awt.event.*;

public class KeyEventExample extends JFrame implements KeyListener {

    private JTextArea textArea;

    public KeyEventExample() {
        setTitle("Key Event Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        textArea = new JTextArea();
        textArea.addKeyListener(this);
```

```
        add(textArea);

        setVisible(true);
    }

    @Override
    public void keyTyped(KeyEvent e) {
        System.out.println("Key Typed: " + e.getKeyChar());
    }

    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("Key Pressed: " +
KeyEvent.getKeyText(e.getKeyCode()));
    }

    @Override
    public void keyReleased(KeyEvent e) {
        System.out.println("Key Released: " +
KeyEvent.getKeyText(e.getKeyCode()));
    }

    public static void main(String[] args) {
        new KeyEventExample();
    }
}
```

Explanation of the Program

- **Imports**: `javax.swing.*` for Swing components, `java.awt.event.*` for event handling.
- **Class**: `KeyEventExample` extends `JFrame` and implements `KeyListener`.
- **Constructor**:
  - Sets up the frame properties (title, size, close operation, and location).
  - Creates a `JTextArea` component and adds a key listener to it.
  - Adds the `JTextArea` to the frame.
- **Key Listener Methods**:
  - `keyTyped()`: Triggered when a key is typed (i.e., a key is pressed and then released).
  - `keyPressed()`: Triggered when a key is pressed.
  - `keyReleased()`: Triggered when a key is released.
- **Main Method**: Creates an instance of the `KeyEventExample` class to display the frame.

6. Write a program to create four threads using Runnable interface.

Here's a Java program that creates four threads using the `Runnable` interface:

```
class MyRunnable implements Runnable {
    private String threadName;

    public MyRunnable(String threadName) {
        this.threadName = threadName;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + " is running: iteration " + i);
            try {
                // Sleep for a while to simulate some work
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted.");
            }
        }
        System.out.println(threadName + " finished.");
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        // Create four instances of MyRunnable, each with a different name
        MyRunnable runnable1 = new MyRunnable("Thread 1");
        MyRunnable runnable2 = new MyRunnable("Thread 2");
        MyRunnable runnable3 = new MyRunnable("Thread 3");
        MyRunnable runnable4 = new MyRunnable("Thread 4");

        // Create four threads and pass the Runnable instances to them
        Thread thread1 = new Thread(runnable1);
        Thread thread2 = new Thread(runnable2);
        Thread thread3 = new Thread(runnable3);
        Thread thread4 = new Thread(runnable4);

        // Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
```

```
    }
}
```

Explanation

1. **MyRunnable Class**: This class implements the `Runnable` interface and defines the `run` method. The `run` method contains the code that the thread will execute. Each thread will print its name and the iteration number, then sleep for 500 milliseconds to simulate some work.
2. **MultiThreadExample Class**: The main class that creates and starts the threads.
    - Four instances of `MyRunnable` are created, each with a unique name.
    - Four `Thread` objects are created, each taking one of the `MyRunnable` instances as its target.
    - The threads are started by calling the `start` method on each `Thread` object.

7. Discuss the differences between HashList and HashMap, Set and List.

Differences between HashList and HashMap

HashMap

- **Definition**: A `HashMap` is a part of the Java Collections Framework and is used to store key-value pairs. It implements the `Map` interface.
- **Key Characteristics**:
    - Allows null keys and values.
    - Unordered: The order of the elements is not guaranteed.
    - Time complexity: O(1) for get and put operations on average.
    - Duplicate keys are not allowed.
- **Use Case**: When you need to store data in key-value pairs and access it quickly by key.

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Alice", 30);
        map.put("Bob", 25);
        map.put("Charlie", 35);
```

```
        System.out.println("Age of Bob: " + map.get("Bob"));
    }
}
```

HashList

- **Definition**: `HashList` is not a standard part of the Java Collections Framework. However, it is typically used to refer to a combination of a list and a hash map, providing both order and fast access.
- **Key Characteristics**:
  - Maintains order like a list.
  - Provides fast access like a hash map.
  - Can be implemented by combining a `HashMap` and a `List`.
- **Use Case**: When you need to maintain the order of elements and also require fast access by key.

```
import java.util.*;

public class HashListExample<K, V> {
    private final List<K> keyOrder = new ArrayList<>();
    private final Map<K, V> map = new HashMap<>();

    public void put(K key, V value) {
        if (!map.containsKey(key)) {
            keyOrder.add(key);
        }
        map.put(key, value);
    }

    public V get(K key) {
        return map.get(key);
    }

    public List<K> keys() {
        return new ArrayList<>(keyOrder);
    }

    public static void main(String[] args) {
        HashListExample<String, Integer> hashList = new HashListExample<>();
        hashList.put("Alice", 30);
        hashList.put("Bob", 25);
        hashList.put("Charlie", 35);

        System.out.println("Keys in order: " + hashList.keys());
        System.out.println("Age of Bob: " + hashList.get("Bob"));
```

```
    }
}
```

Differences between Set and List

Set

- **Definition**: A `Set` is a collection that cannot contain duplicate elements. It is part of the Java Collections Framework and implements the `Set` interface.
- **Key Characteristics**:
  - No duplicate elements.
  - May or may not maintain order (depends on implementation: `HashSet` does not maintain order, `LinkedHashSet` maintains insertion order, `TreeSet` maintains natural ordering).
  - Common implementations: `HashSet`, `LinkedHashSet`, `TreeSet`.
  - No index-based access.
- **Use Case**: When you need to ensure that no duplicate elements are stored.

```
import java.util.HashSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("Alice");
        set.add("Bob");
        set.add("Charlie");
        set.add("Alice"); // Duplicate, will not be added

        System.out.println("Set: " + set);
    }
}
```

List

- **Definition**: A `List` is an ordered collection (also known as a sequence). It is part of the Java Collections Framework and implements the `List` interface.
- **Key Characteristics**:
  - Allows duplicate elements.
  - Maintains insertion order.
  - Index-based access to elements.
  - Common implementations: `ArrayList`, `LinkedList`, `Vector`.

- **Use Case**: When you need to maintain an ordered collection with potential duplicates and require index-based access.

```
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");
        list.add("Alice"); // Duplicate, will be added

        System.out.println("List: " + list);
        System.out.println("Element at index 1: " + list.get(1));
    }
}
```

Differences

Feature	HashMap	HashList (Hypothetical)	Set	List
Definition	Key-value pairs, no order	Combines list order with map access	No duplicates, unordered or ordered	Ordered collection, allows duplicates
Key/Value	Yes	Yes	No	No
Duplicates	Keys: No, Values: Yes	Keys: No, Values: Yes	No	Yes
Order	No	Yes	No (HashSet), Yes (LinkedHashSet, TreeSet)	Yes
Index Access	No	No	No	Yes
Use Case	Fast access by key	Ordered access with fast key lookup	Ensure no duplicates	Maintain order, allow duplicates, index access

8. What are the various components of Swing? Explain

Swing is a part of Java's standard library, specifically designed for creating graphical user interfaces (GUIs). Swing provides a rich set of components that can be used to create sophisticated and interactive user interfaces. Here are some of the key components of Swing and their explanations:

1. Top-Level Containers

These are the primary containers that provide a place for other Swing components to paint themselves.

- **JFrame**: A top-level window with a title and a border. It is commonly used for creating the main window of an application.

```
JFrame frame = new JFrame("My Application");
frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

- **JDialog**: A pop-up window that is typically used for taking input from the user or displaying messages.

```
JDialog dialog = new JDialog(frame, "My Dialog", true);
dialog.setSize(200, 150);
dialog.setVisible(true);
```

- **JApplet**: A component that is used to create applets that can be embedded in web pages.

```
public class MyApplet extends JApplet {
    public void init() {
        add(new JLabel("Hello, Applet!"));
    }
}
```

- **JWindow**: A top-level container without a title, border, or menu bar. It is often used for creating splash screens.

```
JWindow window = new JWindow();
window.setSize(300, 200);
window.setVisible(true);
```

2. Control Components

These are basic user interface controls that allow users to interact with the application.

- **JButton**: A push button that can trigger an action when clicked.

```
JButton button = new JButton("Click Me");
button.addActionListener(e -> System.out.println("Button clicked!"));
```

- **JCheckBox**: A box that can be checked or unchecked to represent a boolean choice.

```
JCheckBox checkBox = new JCheckBox("Check Me");
```

- **JRadioButton**: A button that can be selected or deselected, usually used in a group where only one button can be selected at a time.

```
JRadioButton radioButton = new JRadioButton("Option 1");
```

- **JComboBox**: A drop-down list that allows users to choose one option from a list.

```
JComboBox<String> comboBox = new JComboBox<>(new String[]{"Option 1",
"Option 2"});
```

- **JList**: A component that displays a list of items from which the user can select one or more.

```
JList<String> list = new JList<>(new String[]{"Item 1", "Item 2"});
```

3. Display Components

These components are used to display information to the user.

- **JLabel**: A simple text label.

```
JLabel label = new JLabel("This is a label");
```

- **JTextField**: A single-line text input field.

```
JTextField textField = new JTextField(20);
```

- **JTextArea**: A multi-line text input field.

```
JTextArea textArea = new JTextArea(5, 20);
```

- **JPasswordField**: A single-line text input field that hides the input text, typically used for password input.

```
JPasswordField passwordField = new JPasswordField(20);
```

4. Container Components

These components are used to hold and organize other components.

- **JPanel**: A generic container for grouping components.

```
JPanel panel = new JPanel();
panel.add(new JButton("Button"));
```

- **JScrollPane**: A container that provides a scrollable view of another component.

```
JTextArea textArea = new JTextArea(5, 20);
JScrollPane scrollPane = new JScrollPane(textArea);
```

- **JSplitPane**: A container that divides two components either vertically or horizontally, allowing the user to adjust the size of each component.

```
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                      new JButton("Left"), new
JButton("Right"));
```

- **JTabbedPane**: A container that allows for tabbed navigation between different panels.

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab("Tab 1", new JPanel());
tabbedPane.addTab("Tab 2", new JPanel());
```

5. Advanced Components

These components provide more complex functionalities.

- **JTable**: A component that displays data in a two-dimensional table format.

```
String[] columnNames = {"Name", "Age"};
Object[][] data = {{"John", 25}, {"Jane", 30}};
JTable table = new JTable(data, columnNames);
```

- **JTree**: A component that displays a hierarchical tree of data.

```
JTree tree = new JTree();
```

- **JToolBar**: A container for grouping commonly used actions or tools.

```
JToolBar toolBar = new JToolBar();
toolBar.add(new JButton("New"));
toolBar.add(new JButton("Open"));
```

Program Using Swing Components

```
import javax.swing.*;

public class SwingExample {
    public static void main(String[] args) {
        // Create a new frame
        JFrame frame = new JFrame("Swing Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a panel to hold components
        JPanel panel = new JPanel();

        // Add components to the panel
        panel.add(new JLabel("Enter your name:"));
        panel.add(new JTextField(15));
        panel.add(new JButton("Submit"));

        // Add panel to frame
        frame.add(panel);

        // Display the frame
        frame.setVisible(true);
```

```
    }
}
```