1. Introduction to String Handling

In Java, strings are objects used to store and manipulate sequences of characters. Java provides several classes, such as String, StringBuilder, and StringBuffer, for handling strings. Strings in Java are immutable, meaning once created, their values cannot be changed. This immutability allows for more efficient memory usage and easier handling of strings.

The java.lang.String class is used to create a string object.

There are two ways to create String object:

1. By string literal

2. By new keyword

1. By String Literal

Java String literal is created by using double quotes. For Example:

String s="welcome";

2. By new keyword

String s=new String("Welcome");

String methods in java

1. length(): Returns the number of characters in the string.

```
String str = "Hello, World!";
int len = str.length(); // 13
System.out.println("Length of the string: " + len);
```

Output:

Length of the string: 13

2. equals(): Checks if two strings have the same content (case-sensitive).

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = "hello";
boolean isEqual = str1.equals(str2); // true
boolean isEqualCaseSensitive = str1.equals(str3); // false
System.out.println("str1 equals str2: " + isEqual);
System.out.println("str1 equals str3 (case-sensitive): " + isEqualCaseSensitive);
```

Output:

str1 equals str2: true

str1 equals str3 (case-sensitive): false

3. equalsIgnoreCase(): Compares strings, ignoring case differences.

```
String str1 = "Hello";
String str2 = "hello";
boolean isEqualIgnoreCase = str1.equalsIgnoreCase(str2); // true
System.out.println("str1 equals str2 (ignoring case): " + isEqualIgnoreCase);
```

Output:

str1 equals str2 (ignoring case): true

4. startsWith(String prefix): Checks if the string starts with the specified prefix.

```
String str = "Java Programming";
boolean startsWithJava = str.startsWith("Java"); // true
System.out.println("String starts with 'Java': " + startsWithJava);
```

Output:

String starts with 'Java': true

5. endsWith(String suffix): Checks if the string ends with the specified suffix.

```
String str = "Hello, World!";
boolean endsWithWorld = str.endsWith("World!"); // true
System.out.println("String ends with 'World!': " + endsWithWorld);
```

Output:

String ends with 'World!': true

6. StringBuffer reverse()

Reverses the contents of a StringBuffer.

```
StringBuffer sb = new StringBuffer("Hello");
sb.reverse(); // "olleH"
System.out.println("Reversed StringBuffer: " + sb);
```

Output:

Reversed StringBuffer: olleH

7. replace(char oldChar, char newChar): Replaces all occurrences of a specified character in a string.

```
String str = "balloon";
String replacedStr = str.replace('o', 'a'); // "ballaan"
System.out.println("Replaced String: " + replacedStr);
```

Output:

Replaced String: ballaan

8. concat(String str): Concatenates the specified string to the end of the current string.

```
String str1 = "Hello";
String str2 = str1.concat(" World");
System.out.println("Concatenated String: " + str2);
```

Output:

Concatenated String: Hello World

9.charAt(int index): Returns the character at the specified index.

```
String str = "Hello, World!";
char ch = str.charAt(7); // 'W'
System.out.println("Character at index 7: " + ch);
```

Output:

Character at index 7: World

10. substring(int start, int end): Returns a new string containing the characters from the specified start to end index.

```
String str = "Hello, World!";
String subStr = str.substring(7, 12); // "World"
System.out.println("Substring from index 7 to 12: " + subStr);
```

Output:

Substring from index 7 to 12: World

11.toCharArray(): Converts the string to a character array.

```
String str = "Hello";
char[] charArray = str.toCharArray();
System.out.println("Character array: " + Arrays.toString(charArray));
```

Output: Character array: [H, e, l, l, o]

12. compareTo(String anotherString): Compares two strings lexicographically.

```
String str1 = "Apple";
String str2 = "Banana";
int comparison = str1.compareTo(str2); // returns a negative value because "Apple" < "Banana"
System.out.println("Comparison result: " + comparison);
```

Output:

Comparison result: -1

13. concat(String str): Concatenates the specified string to the end of the current string.

```
String str1 = "Hello";
String str2 = “world”
System.out.println(str1.concat(str2);
```

Output:

Concatenated String: Hello World

14. replaceAll(String regex, String replacement): Replaces each substring that matches the given regular expression with the specified replacement.

```
String sentence = "The rain in Spain";

String replacedSentence = sentence.replaceAll("ain", "oon"); // "The roon in Spoon"

System.out.println("Replaced Sentence: " + replacedSentence);
```

Output:

Replaced Sentence: The roon in Spoon

15.toLowerCase() and toUpperCase(): Converts all characters in the string to lowercase or uppercase.

```
String str = "Hello, World!";

String lower = str.toLowerCase(); // "hello, world!"

String upper = str.toUpperCase(); // "HELLO, WORLD!"

System.out.println("Lowercase: " + lower);

System.out.println("Uppercase: " + upper);
```

Output:

Lowercase: hello, world!

Uppercase: HELLO, WORLD!

Multithreading

Multithreaded programming is a method of concurrent execution in which multiple threads, or smaller units of a process, run simultaneously. This technique enhances the efficiency of a program, particularly on multi-core processors, by allowing multiple tasks to execute at once. Let's break down some of the essential concepts in multithreaded programming:

1. Need for Multiple Threads

Multiple threads enable concurrent execution, which improves program performance and responsiveness. For example, in a GUI application, one thread can handle the user interface while another thread performs calculations in the background.

2. Multithreaded Programming for Multi-core Processors

Multi-core processors can execute multiple threads in parallel, allowing programs to take full advantage of the processor's capabilities. This enables faster computation and can reduce the time required for processing tasks.

Thread Life Cycle

During the life time of a thread there are many states it can enter. They are

A. NewBorn state

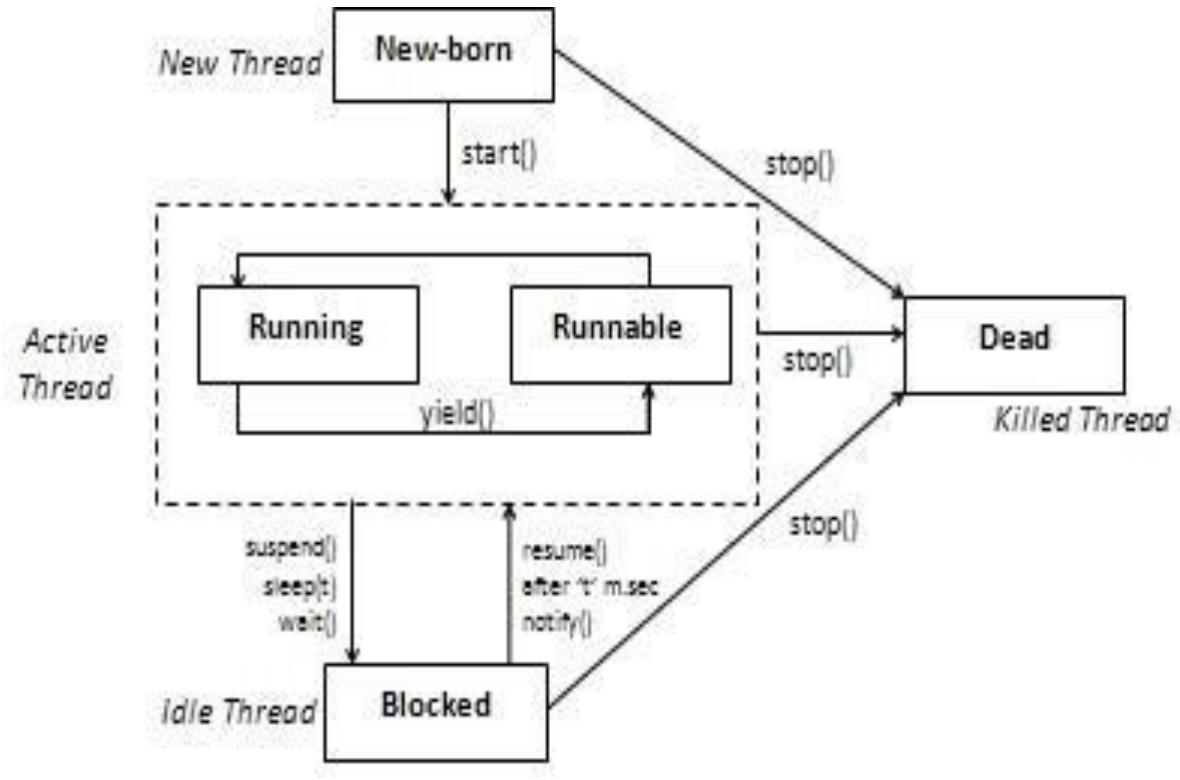
B. Runnable state

C. Running State

D. Blocked state

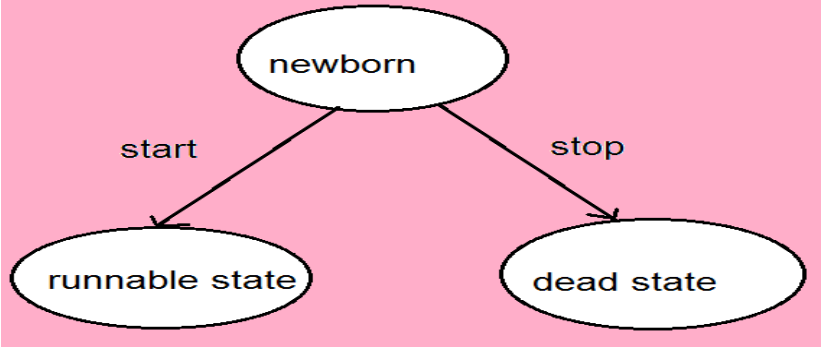
E. Dead state

A thread is always in any one of these five states.It can move from one state to another via a variety of ways as shown below



New Born state

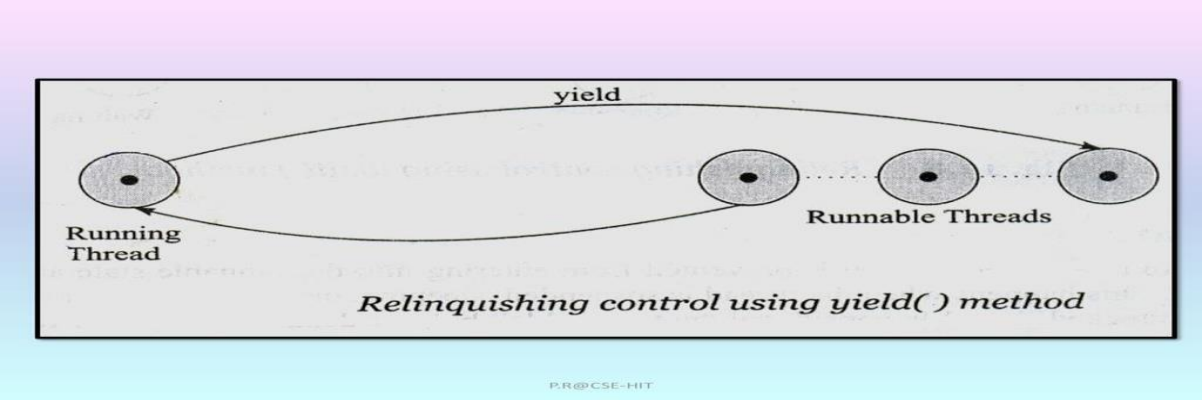
- ✓ when we create a thread object, the thread is born and is said to be in new born state.
- ✓ The thread is not yet scheduled for running .At this state, we can do only one of the following things with it.
- ❖ Schedule it for running using start() method.
- ❖ Kill it using stop() method.



- ❖ **If scheduled ,it moves to the runnable state**

Runnable State

- ➢ The runnable state means that the thread is ready for execution and is waiting for availability of the processor .i.e the thread has joined the queue of threads that are waiting for execution.
- ➢ If all threads have equal priority, then they are given time slots for execution in Round Robin fashion,i.e FCFS manner.
- ➢ The thread that relinguishes control joins the queue at the end & again waits for its turn



Running State

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it relinguishes its control in one of the following situations.

1) It has been suspended using suspend().

   a suspended thread can be received by using the resume() method.

2) It has been made wait by using wait() method

   A thread that is waiting will get resumed after notify() method

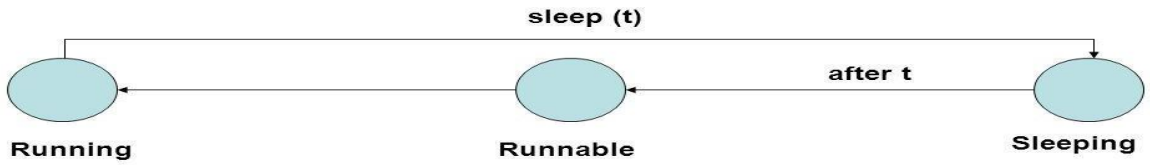
3) It has been slept for a t seconds.

A thread will get invoked after t seconds

Example

2. sleep()

It has been made to sleep, We can put a thread to sleep for a specified time period using the method sleep (time) where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re – enters the runnable state as soon as this time period is elapsed.



Blocked state

- Thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods.

Sleep() **// blocked for specified time**

Suspended() **// blocked until further orders**

Wait() **// blocked until certain condition occurs**.

- These methods cause the thread to go into the blocked sate. The thread will return to runnable state when the specified time is elapsed in the case of sleep(),the resume() method is invoked in case of suspend(),and notify() method is called in case of wait().

Dead State

- Every thread has a life cycle .
- A running thread ends its life when it has completed executing its run().it is natural death.
- However, we can kill it by using stop message to it at any stage.Thus causing premature death to it.

Creating threads in java is simple. Threads in java can be created in two ways

1) By extending the thread class.

2) By implementing the runnable interface.

1) **Creating threads by extending the thread class:**

- Define a class that extends thread class and override its run()with the code required by the thread.
- Steps to create thread by extending thread class are
    a) Declaring the class
    b) Implementing the run() method.
    c) Starting New Thread.

Declaring the class:

Declare the class by extending the thread

class as: Class MyThread extends Thread

```
{
   -------
   -------
}
```

Implementing the run() method:

- the run method is the heart and soul of any thread.
- We have to override this method in order to implement the code to be executed by our thread.
- It makes up the entire body of a thread and is the only method in which the threads behavior can be implemented.
- The basic implementation of run()

  will look like public void run()

```
{
Thread code
}
```

When we start new thread ,java calls the threads run() method.

Starting New Thread:

- create a thread object and call the start() method to initiate the thread execution.
- To create and run an instance of our thread class, we must write the

following: MyThread t1=new MyThread();

T1.start();

- The first line instantiates a new object of class MyThread.
- The second line calls start() causing the thread to move into runnable state.
- Then, the java runtime will schedule the thread to run by invoking its run().Hence the thread is said to be in Running state.

```
class A extends Thread
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread A :"+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
class B extends Thread
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread B :"+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
public class HelloWorld{

     public static void main(String []args)
     {
        A ob1 = new A();
        B ob2 = new B();
        ob1.start();
        ob2.start();
     }
}
```

```
$javac HelloWorld.java
$java -Xmx128M -Xms16M HelloWorld
From Thread A :1
From Thread B :1
From Thread A :2
From Thread B :2
From Thread A :3
From Thread B :3
From Thread A :4
From Thread B :4
From Thread A :5
From Thread B :5
```

Creating Thread using Runnable Interface

A) Create a Class that implements Runnable Interface

B) override run() method

create a thread by passing an object to the implementation class of runnable interface

```
class A implements Runnable
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread A: "+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
class B implements Runnable
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread B: "+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
public class HelloWorld{

     public static void main(String []args)
     {
        A ob1 = new A();
        B ob2 = new B();
        Thread t1 = new Thread(ob1);
        Thread t2 = new Thread(ob2);
        t1.start();
        t2.start();
     }
}
```

```
$javac HelloWorld.java
$java -Xmx128M -Xms16M HelloWorld
From Thread B: 1
From Thread A: 1
From Thread B: 2
From Thread A: 2
From Thread B: 3
From Thread A: 3
From Thread B: 4
From Thread A: 4
From Thread B: 5
From Thread A: 5
```

Thread Priority and Synchronization

Thread Priority is a concept in multithreaded programming that determines the relative importance of each thread when they compete for CPU time. Thread priorities help the system's scheduler decide which thread to run when multiple threads are ready for execution.

1. **Priority Levels**:
    - Threads are assigned priority levels, typically as integers. In Java, for example, thread priorities range from MIN_PRIORITY (1) to MAX_PRIORITY (10), with NORM_PRIORITY (5) as the default.
    - A higher priority thread is more likely to be selected by the CPU scheduler over a lower-priority thread, although this behavior is platform-dependent.

Synchronization in multithreaded programming is crucial for managing access to shared resources to avoid data inconsistency and ensure thread safety.

1. **The Need for Synchronization**:
    - When multiple threads access shared resources (e.g., shared variables, files, or memory), there is a risk of **race conditions**, where the final outcome depends on the timing of thread execution.
    - Synchronization prevents threads from interfering with each other and ensures that only one thread accesses a shared resource at a time.
2. **Synchronized Blocks and Methods**:
    - In many programming languages, synchronization is achieved using synchronized blocks or methods. A synchronized block allows only one thread at a time to access the code block or resource.
    - For example, in Java, the synchronized keyword locks an object, so no other thread can access the synchronized code block or method of that object until the current thread completes it.
3. **Locks (Mutexes)**:
    - A **lock** (or **mutex**) is a mechanism used to enforce synchronization by allowing only one thread to hold the lock at a time.
    - When a thread acquires a lock on a resource, other threads must wait until the lock is released before they can access the same resource.
4. **Deadlock**:
    - **Deadlock** occurs when two or more threads wait indefinitely for resources held by each other, creating a cycle of dependencies with no resolution.
    - Avoiding deadlock requires careful resource allocation and sometimes the use of timeout-based locking mechanisms.

5. **Avoiding Race Conditions**:
    - Race conditions occur when multiple threads attempt to modify shared data concurrently, leading to inconsistent results. Synchronization helps avoid race conditions by enforcing an orderly access to shared resources.

Deadlock and RaceConditions

Both deadlock and race conditions are critical concurrency issues in multithreaded programming.

Deadlock involves threads waiting indefinitely for each other, which halts progress, often requiring a restart or intervention.

Race Conditions involve unpredictable results due to concurrent access to shared data, leading to data inconsistency.

Using synchronization techniques and careful resource management can help prevent both deadlock and race conditions, resulting in safer and more predictable multithreaded programs.

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a standard Java API that enables Java applications to interact with a wide range of databases. It provides methods for querying and updating data in a database and is widely used for developing Java applications that need to communicate with databases like MySQL, PostgreSQL, Oracle, and others.

1. Introduction to JDBC

JDBC allows Java programs to:

- Connect to a database.
- Send SQL queries and update statements to the database.
- Process the results retrieved from the database.

JDBC provides a universal data access API that is independent of any particular database or platform, enabling developers to switch databases without altering their Java code significantly.

2. JDBC Architecture

The JDBC architecture consists of two main components:

1. **JDBC API**: This provides a standard interface for Java applications to connect to the database, execute SQL queries, and retrieve results. The JDBC API includes classes and interfaces such as DriverManager, Connection, Statement, PreparedStatement, and ResultSet.
2. **JDBC Driver**: JDBC drivers are database-specific implementations of the JDBC API that communicate with the database. JDBC drivers translate the API calls into database-specific calls, making the interaction between Java applications and databases possible. There are four types of JDBC drivers:
    - **Type 1**: JDBC-ODBC Bridge Driver
    - **Type 2**: Native API Driver
    - **Type 3**: Network Protocol Driver
    - **Type 4**: Thin Driver (pure Java driver; commonly used for databases like MySQL)

3. Installing MySQL and MySQL Connector/J

To use JDBC with MySQL, you need to install both MySQL and the MySQL Connector/J.

Installing MySQL

1. Download the MySQL installer from the MySQL official website.
2. Run the installer and follow the installation steps.
3. Set up a root password and configure any other settings as needed.

Installing MySQL Connector/J

The MySQL Connector/J is the JDBC driver for MySQL, which is required to connect Java applications to a MySQL database.

1. Download the MySQL Connector/J from the MySQL Connector/J download page.
2. Extract the downloaded ZIP file, and locate the mysql-connector-java-<version>.jar file.
3. Add this .jar file to your project's classpath. In IDEs like IntelliJ or Eclipse, you can do this by right-clicking your project, selecting "Add External JARs," and choosing the Connector/J JAR file.

4. JDBC Environment Setup

To set up the JDBC environment in a Java application:

1. Ensure the MySQL Connector/J JAR file is in your project's classpath.
2. Import necessary JDBC packages:

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.SQLException;
```

5. Establishing JDBC Database Connections

To establish a connection with a database in Java, follow these steps:

1. **Load the JDBC Driver** (optional since JDBC 4.0):

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

2. **Establish a Connection**:
    - Use DriverManager.getConnection() with the JDBC URL, username, and password.
    - The JDBC URL format for MySQL is:

```
jdbc:mysql://<hostname>:<port>/<database_name>
```

For example:

```
String url = "jdbc:mysql://localhost:3306/mydatabase";

String username = "root";

String password = "password";


Connection connection = DriverManager.getConnection(url, username, password);
```

3. **Create a Statement**:

Statement statement = connection.createStatement();

4. **Execute Queries**:

ResultSet resultSet = statement.executeQuery("SELECT * FROM users");

5. **Close the Connection**:

resultSet.close();

statement.close();

connection.close();

6. ResultSet Interface

The ResultSet interface represents the result set obtained by executing a SQL query and provides methods to navigate and retrieve data from it. A ResultSet can be thought of as a table of data, with rows representing each record returned by the query.

Commonly Used Methods of the ResultSet Interface

1. **Navigating the ResultSet**:
    - next(): Moves the cursor to the next row. Returns false if there are no more rows.
    - previous(): Moves the cursor to the previous row (only if ResultSet is scrollable).
    - first(), last(): Moves to the first or last row.
2. **Retrieving Data**:
    - getString(columnLabel): Retrieves a column as a String.
    - getInt(columnLabel): Retrieves a column as an int.
    - getDouble(columnLabel): Retrieves a column as a double.
    - Column labels can be the column name or the column index.

Example Program Using JDBC to Query MySQL Database

Here's an example program that connects to a MySQL database, retrieves data from a table, and displays it.

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.SQLException;

```
public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";

        try (Connection connection = DriverManager.getConnection(url, username, password)) {
            System.out.println("Database connected successfully!");

            Statement statement = connection.createStatement();
            String query = "SELECT id, name, email FROM users";
            ResultSet resultSet = statement.executeQuery(query);

            System.out.println("User Details:");
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");

                System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
            }

            resultSet.close();
            statement.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

This program:

- Connects to a MySQL database.
- Queries a table named users.
- Iterates over the ResultSet to print each user’s id, name, and email.

JavaFX Scene Builder

JavaFX Scene Builder is a **visual design tool** used for building the user interface (UI) of JavaFX applications without manual coding. Instead of writing Java code for UI layouts, Scene Builder allows you to visually design the interface and automatically generates an FXML file to represent the structure.

Key Features

1. **Drag-and-Drop Interface**
    - You can easily add UI components like buttons, labels, and text fields by dragging them from the toolbox onto the design canvas.
2. **Set Properties for Controls**
    - Configure UI components by setting properties such as text, size, alignment, and style directly in Scene Builder.
3. **FXML Code Generation**
    - Automatically generates an FXML file based on the layout you design. This file can be loaded in your JavaFX application for rendering.
4. **Link to Controller Classes**
    - Allows you to assign event handlers and bind UI components to your JavaFX application's controller class.

How to Use

1. **Download and Install Scene Builder**
    - Download Scene Builder from the official Gluon website.
    - Install it on your computer.
2. **Design the User Interface**
    - Open Scene Builder and start a new design.
    - Add nodes like buttons, text fields, or labels by dragging them from the toolbox to the design area.
    - Arrange and configure properties for each node using the Properties panel.
3. **Save as FXML**
    - Once the design is complete, save it as an .fxml file.
    - Example: MainUI.fxml.

4. **Integrate FXML with Your JavaFX Application**
    - Use the FXMLLoader class in your JavaFX code to load the saved FXML file.

Example Code Integration

FXML File (MainUI.fxml):

xml

Copy code

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.StackPane?>

<StackPane xmlns:fx="http://javafx.com/fxml">
   <Button text="Click Me!" fx:id="myButton"/>
</StackPane>
```

Java Application:

java

Copy code

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MainApp extends Application {
   @Override
   public void start(Stage primaryStage) throws Exception {
      Parent root = FXMLLoader.load(getClass().getResource("MainUI.fxml"));
      Scene scene = new Scene(root, 400, 300);
```

```
        primaryStage.setTitle("JavaFX with Scene Builder");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

2. JavaFX App Window Structure

A JavaFX application follows a hierarchical structure where components are organized into three main layers: **Stage**, **Scene**, and **Nodes**. Let's explore these components in detail:

1. Stage

- **What is it?**
  The **Stage** is the top-level container that represents the application window. It is automatically created when a JavaFX application starts.
- **Key Features:**
  - Controls the window's title, size, and visibility.
  - Acts as the main entry point for displaying the user interface.

Example:

```
primaryStage.setTitle("My JavaFX Application");
primaryStage.setWidth(800);
primaryStage.setHeight(600);
```

2. Scene

- **What is it?**
  The **Scene** holds all the visual elements (nodes) of the application and represents the content to be displayed in the Stage.
- **Key Features:**
  - Acts as a container for the **Scene Graph**, which is a hierarchical tree of nodes.

- Defines properties like dimensions and styling.
- A Stage can have only one Scene at a time, but the Scene can be swapped dynamically.

Example:

```
Scene scene = new Scene(rootNode, 400, 300);

primaryStage.setScene(scene);
```

3. Nodes

- **What are they?**
  Nodes are the building blocks of the Scene Graph. They are individual components like buttons, labels, text fields, and layout panes.
- **Types of Nodes:**
  - **Root Node:** The top-most node in the Scene Graph (e.g., layout panes like StackPane, VBox, etc.).
  - **Child Nodes:** UI elements (e.g., Button, Label, Text, ImageView) added to the Root Node or other containers.

Example:

```
Label label = new Label("Hello, JavaFX!");

Button button = new Button("Click Me");

VBox rootNode = new VBox(10, label, button);
```

Complete Example:

Here's a simple JavaFX application demonstrating the structure:

```
import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;


public class JavaFXAppStructure extends Application {

   @Override
```

```
    public void start(Stage primaryStage) {
        // Create Nodes
        Label label = new Label("Welcome to JavaFX!");
        Button button = new Button("Click Me");

        // Create Root Node (Layout Pane)
        VBox rootNode = new VBox(10, label, button);

        // Create Scene and Set Dimensions
        Scene scene = new Scene(rootNode, 400, 300);

        // Set Scene to the Stage
        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX App Window Structure");
        primaryStage.show();
    }

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

3. Displaying Text and Images in JavaFX

JavaFX provides straightforward ways to display both **text** and **images** in a user interface. Here's how you can use the Label, Text, and ImageView nodes effectively:

Displaying Text

Options:

1. **Label**
    - Used for short, non-editable text.
    - Often used in forms or as a description for UI components.

Example:

```
Label label = new Label("Welcome to JavaFX!");
```

2. **Text**
    - More flexible than Label, allowing custom fonts, styles, and multi-line text.
    - Used for rich text display or larger content.

Example:

```
Text text = new Text("Hello, JavaFX Text Node!");
text.setStyle("-fx-font-size: 20px; -fx-fill: blue;");
```

Code Example for Text Display:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class DisplayTextExample extends Application {
   @Override
   public void start(Stage primaryStage) {
      Label label = new Label("This is a Label!");
      Text text = new Text("This is a Text node!");

      VBox root = new VBox(10, label, text);

      Scene scene = new Scene(root, 300, 200);
      primaryStage.setTitle("Displaying Text");
      primaryStage.setScene(scene);
      primaryStage.show();
   }
```

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

Displaying Images

Using Image and ImageView:

1. **Image:** Represents the image file loaded from a URL or local file.
2. **ImageView:** Displays the image in the scene.

Steps:

- Create an Image object.
- Pass it to an ImageView.

Code Example for Image Display:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class DisplayImageExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Load image from a file (adjust the file path as needed)
        Image image = new Image("file:your_image_path.jpg");
        ImageView imageView = new ImageView(image);

        // Optional: Set image dimensions
        imageView.setFitWidth(200);
        imageView.setPreserveRatio(true);
```

```
        VBox root = new VBox(imageView);

        Scene scene = new Scene(root, 300, 300);
        primaryStage.setTitle("Displaying Image");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

4. Event Handling in JavaFX

Event handling in JavaFX allows you to define actions or behaviors that occur when a user interacts with UI components, such as clicking a button, pressing a key, or moving the mouse. JavaFX uses an event-driven model to handle these interactions.

Key Components of Event Handling

1. **Event Source**
   The UI component that generates the event (e.g., Button, TextField).
2. **Event Handler**
   A method or lambda expression that defines the response to the event.
3. **Event Object**
   Provides information about the event, such as the source of the event and event type.

Steps to Handle Events

1. Set an Event Handler

You can set an event handler for a UI component using:

- A **Lambda Expression**
- An **Anonymous Class**
- A **Separate Method**

2. Use Event Methods

The most common method for handling events is setOnAction, which is used for buttons and similar controls.

Examples

1. Button Click Event

Using a **Lambda Expression**:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ButtonEventExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me");
        button.setOnAction(e -> System.out.println("Button clicked!"));

        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Button Click Event");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

2. Handling Mouse Events

JavaFX provides methods like setOnMouseEntered and setOnMouseClicked for handling mouse interactions.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MouseEventExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Text text = new Text("Hover over me!");
        text.setOnMouseEntered(e -> text.setText("Mouse Entered!"));
        text.setOnMouseExited(e -> text.setText("Hover over me!"));

        StackPane root = new StackPane(text);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Mouse Event Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

3. Handling Events with a Separate Method

You can define a separate method to handle the event.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class SeparateMethodEventExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me");
        button.setOnAction(this::handleButtonClick);

        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Event Handling with Separate Method");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private void handleButtonClick(javafx.event.ActionEvent event) {
        System.out.println("Button was clicked!");
    }

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

}

Event Types

- **ActionEvent**: Triggered by actions like button clicks or menu item selection.
- **MouseEvent**: Triggered by mouse actions like clicks or movement.
- **KeyEvent**: Triggered by keyboard actions like key presses or releases.
- **WindowEvent**: Triggered by changes in the application window (e.g., close or resize).

---

5. Laying Out Nodes in the Scene Graph

In JavaFX, layout panes are used to organize and position nodes (UI components) within the Scene Graph. Each layout pane provides a specific way to arrange its children.

Common Layout Panes

1. HBox (Horizontal Layout)

- **Description:** Arranges its children in a single horizontal row.
- **Use Case:** Useful for toolbars or placing buttons side-by-side.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class HBoxExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");

        HBox hbox = new HBox(10, btn1, btn2, btn3); // Spacing between nodes
```

```
        Scene scene = new Scene(hbox, 300, 100);

        primaryStage.setTitle("HBox Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

2. VBox (Vertical Layout)

- **Description:** Arranges its children in a single vertical column.
- **Use Case:** Useful for forms, menus, or stacked controls.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class VBoxExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");

        VBox vbox = new VBox(10, btn1, btn2, btn3); // Spacing between nodes
```

```
        Scene scene = new Scene(vbox, 200, 150);

        primaryStage.setTitle("VBox Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

3. GridPane (Grid Layout)

- **Description:** Arranges children in a flexible grid of rows and columns.
- **Use Case:** Useful for complex forms or tables.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class GridPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");
        Button btn4 = new Button("Button 4");
```

```
        GridPane grid = new GridPane();
        grid.setHgap(10); // Horizontal gap between columns
        grid.setVgap(10); // Vertical gap between rows

        // Adding buttons to the grid (column, row)
        grid.add(btn1, 0, 0);
        grid.add(btn2, 1, 0);
        grid.add(btn3, 0, 1);
        grid.add(btn4, 1, 1);

        Scene scene = new Scene(grid, 300, 200);
        primaryStage.setTitle("GridPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

4. BorderPane (Border Layout)

- **Description:** Divides the layout into five regions: top, bottom, left, right, and center.
- **Use Case:** Useful for creating applications with a header, footer, sidebar, and main content.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
```

```
public class BorderPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button topButton = new Button("Top");
        Button bottomButton = new Button("Bottom");
        Button leftButton = new Button("Left");
        Button rightButton = new Button("Right");
        Button centerButton = new Button("Center");

        BorderPane borderPane = new BorderPane();
        borderPane.setTop(topButton);
        borderPane.setBottom(bottomButton);
        borderPane.setLeft(leftButton);
        borderPane.setRight(rightButton);
        borderPane.setCenter(centerButton);

        Scene scene = new Scene(borderPane, 400, 300);
        primaryStage.setTitle("BorderPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

6. Handling Mouse Events in JavaFX

JavaFX provides a rich set of mouse events to handle interactions such as clicks, drags, and hover actions. These events are defined in the MouseEvent class, and you can attach event handlers to any node in your scene.

Common Mouse Events

1. **Mouse Click Events**
    - setOnMouseClicked: Triggered when a mouse button is clicked on a node.
2. **Mouse Hover Events**
    - setOnMouseEntered: Triggered when the mouse enters a node.
    - setOnMouseExited: Triggered when the mouse leaves a node.
3. **Mouse Drag Events**
    - setOnMouseDragged: Triggered when the mouse is dragged while pressing a button.
    - setOnMousePressed / setOnMouseReleased: Triggered when the mouse button is pressed/released.

Example 1: Handling a Mouse Click

This example changes the text of a Label when clicked.

```
import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Label;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;


public class MouseClickExample extends Application {

   @Override

   public void start(Stage primaryStage) {

      Label label = new Label("Click Me!");


      // Set Mouse Click Event

      label.setOnMouseClicked(e -> label.setText("Label Clicked!"));
```

```
        StackPane root = new StackPane(label);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Mouse Click Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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