**LAB # 11**

**Threads – Multiple Threads**

In Android, threads and multithreading refer to the concept of running multiple threads of execution concurrently within an application.

A thread is a lightweight unit of execution that can run independently and perform tasks in parallel. By default, Android applications run on a single main thread called the "UI thread" or "main thread." This thread is responsible for handling user interface updates and responding to user interactions. However, performing long-running or resource-intensive tasks on the main thread can cause the UI to freeze, resulting in an unresponsive user interface.

Multithreading, on the other hand, involves creating and managing multiple threads to execute tasks concurrently. It allows you to divide a complex task into smaller subtasks that can be executed simultaneously on separate threads, thereby improving performance and responsiveness of your application.

In Android, multithreading is commonly used for performing tasks such as network operations, database operations, file I/O, image processing, or any other CPU-intensive or time-consuming tasks that should not block the main thread.

By using multithreading, you can offload these tasks to background threads, freeing up the main thread to focus on handling UI updates and user interactions. This helps to ensure a smooth and responsive user interface, preventing the app from becoming unresponsive or exhibiting a sluggish behavior.

Android provides several mechanisms to work with multithreading, including**:**

**1. Java Threads:** You can create threads in Java using the `Thread` class or by implementing the `Runnable` interface**.**

**2. `AsyncTask`:** `AsyncTask` is a utility class provided by Android that simplifies working with threads for performing background tasks and updating the UI. It allows you to execute tasks asynchronously and provides convenient methods for updating UI elements.

**3. `Handler` and `Looper`:** The `Handler` class allows you to communicate between different threads by sending messages or posting runnable objects. It works in conjunction with the `Looper` class, which manages the message loop for a thread.

**4. `ExecutorService` and thread pools:** The `ExecutorService` framework provides a higher-level way of managing threads, including thread pools. It simplifies the execution of tasks in the background and offers features such as thread reuse, task scheduling, and thread pooling.

When working with multithreading in Android, it's important to consider thread synchronization, thread safety, and potential race conditions to ensure proper coordination and synchronization between threads.

Overall, multithreading in Android allows you to perform complex tasks efficiently, improve app responsiveness, and deliver a smoother user experience by leveraging the power of concurrent execution on multiple threads.  
  
**Step by Step Android multithreading with multiple threads using Java:**

**Step 1: Create a new Android project**

Start by creating a new Android project in your preferred IDE, such as Android Studio. Set up the project with the necessary configuration, including the minimum SDK version and other project details.

**Step 2: UI Setup**

Design your app's user interface (UI) according to your requirements. For this tutorial, let's assume you have a simple layout with a button to start the background tasks and a text view to display the results.

**Step 3: Add Click Listener to Button**

In your main activity class, retrieve references to the UI elements (button and text view) and set a click listener for the button.

**public class MainActivity extends AppCompatActivity {**

**private Button startButton;**

**private TextView resultTextView;**

**@Override**

**protected void onCreate(Bundle savedInstanceState) {**

**super.onCreate(savedInstanceState);**

**setContentView(R.layout.activity\_main);**

**startButton = findViewById(R.id.start\_button);**

**resultTextView = findViewById(R.id.result\_textview);**

**startButton.setOnClickListener(new View.OnClickListener() {**

**@Override**

**public void onClick(View view) {**

**startBackgroundTasks();**

**}**

**});**

**}**

**private void startBackgroundTasks() {**

**// Start the background tasks here**

**}**

**}**

**Step 4: Implement Background Tasks with Multiple Threads**

Create a new class for each background task that extends the `Thread` class. Each class will represent a separate thread that performs a specific task.

**public class BackgroundTask1 extends Thread {**

**@Override**

**public void run() {**

**// Perform background task 1 here**

**}**

**}**

**public class BackgroundTask2 extends Thread {**

**@Override**

**public void run() {**

**// Perform background task 2 here**

**}**

**}**

**```**

**Step 5: Perform Background Tasks**

**Inside the `startBackgroundTasks()` method, create instances of the background task classes and start them by calling the `start()` method.**

**```java**

**private void startBackgroundTasks() {**

**BackgroundTask1 backgroundTask1 = new BackgroundTask1();**

**BackgroundTask2 backgroundTask2 = new BackgroundTask2();**

**backgroundTask1.start();**

**backgroundTask2.start();**

**}**

**```**

**Step 6: Update UI from Background Threads**

**To update the UI from each background thread, you can use the `runOnUiThread()` method of the `Activity` class. Inside the `runOnUiThread()` method, you can update the UI elements with the results obtained from the background tasks.**

**```java**

**public class BackgroundTask1 extends Thread {**

**@Override**

**public void run() {**

**// Perform background task 1 here**

**// Update UI with the result of task 1**

**MainActivity.this.runOnUiThread(new Runnable() {**

**@Override**

**public void run() {**

**resultTextView.append("Task 1 completed\n");**

**}**

**});**

**}**

**}**

**public class BackgroundTask2 extends Thread {**

**@Override**

**public void run() {**

**// Perform background task 2 here**

**// Update UI with the result of task 2**

**MainActivity.this.runOnUiThread(new Runnable() {**

**@Override**

**public void run() {**

**resultTextView.append("Task 2 completed\n");**

**}**

**});**

**}**

**}**

**Step 7: Test the App**

Now, run the application on an Android device or emulator. When you click the start button, both background tasks will start running concurrently, and their results will be displayed in the text view as they complete.

to handle any synchronization or thread safety issues that may arise in more complex scenarios.

Note: In practice, you may also consider using other concurrency mechanisms such as `ExecutorService`, Kotlin coroutines, or `AsyncTask` based on your specific requirements and the complexity of your app.

**TASK-1:**

You have to create an Android application on which user can create multiple threads by pressing a button multiple time. Each thread must be represented by a progress bar having total 100 steps. Cost of each step is assumed to be 500ms.

You may have the following UI for this task:

