BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 6

DCA3201

MOBILE APPLICATION DEVELOPMENT

Unit 3

Understanding Threads

Table of Contents

1. INTRODUCTION

Multithreading is a fundamental concept in Android app development that lies at the heart of creating responsive and efficient mobile applications. Android devices are powered by multi-core processors, and to harness their full potential, developers need to grasp the intricacies of threads. Threads are essentially small units of execution that allow different parts of an app to run concurrently. In the Android ecosystem, one of the most crucial threads is the Main Thread, also known as the UI Thread, which is responsible for managing the user interface and ensuring a seamless user experience.

However, as Android apps often involve resource-intensive tasks, understanding how to work with multiple threads, including the Main Thread, is essential to avoid application freezes, improve responsiveness, and deliver a smooth user experience.

1.1 Learning Objectives

At the end of the topic, students should be able to:

- *Identify the basic concept of threading and its relevance in Android development.*
- *Explain the benefits and challenges of multithreading in Android.*
- *Describe the significance of the Main Thread in handling UI events.*
- *Assess thread safety considerations and describe strategies for ensuring thread safety in Android development.*
- *Implement AsyncTask's three main methods in a practical Android application.*

2. UNDERSTANDING THREADS IN ANDROID

2.1 Introduction To Multithreading:

Multithreading is a programming technique that allows an application to execute multiple threads concurrently, where each thread represents an independent flow of control within the application. These threads can execute different tasks simultaneously, sharing the resources of a single process. In the context of mobile application development, such as Android, multithreading enables an app to perform multiple operations simultaneously, improving responsiveness and overall performance.

Why Multithreading is Important in Android App Development:

Multithreading is of paramount importance in Android app development for several reasons:

- Responsiveness: Android apps often involve tasks that can take a significant amount of time, such as downloading data from the internet or performing complex calculations. If these tasks were executed on the main thread (also known as the UI thread), it would lead to a sluggish and unresponsive user interface. Multithreading allows these time-consuming tasks to be performed in the background, keeping the UI responsive.
- Efficiency: Mobile devices, including Android smartphones and tablets, are equipped with multi-core processors. Utilizing multiple threads allows developers to leverage the full potential of these processors, ensuring that tasks are executed efficiently and without overloading a single core.
- User Experience: Users expect smooth and uninterrupted interactions with mobile apps. Multithreading helps in achieving this by preventing long-running tasks from blocking the user interface. A responsive app leads to a better user experience and higher user satisfaction.

2.2 Benefits and Challenges of Multithreading:

Benefits:

- Improved Performance: Multithreading can significantly improve the performance of an Android app by parallelizing tasks and utilizing available processor cores effectively.

- Responsiveness: By offloading time-consuming tasks to background threads, multithreading ensures that the UI remains responsive, preventing the app from freezing or becoming unresponsive.
- Resource Utilization: Multithreading allows apps to make better use of system resources, reducing the time it takes to complete tasks and conserving battery life.

Challenges:

- Synchronisation: Managing shared resources between multiple threads can be complex and prone to synchronisation issues, such as race conditions and deadlocks.
- Thread Safety: Ensuring that data is accessed and modified safely by multiple threads requires careful consideration and often involves synchronisation mechanisms.
- Complexity: Multithreaded code can be more challenging to design, debug, and maintain compared to single-threaded code, as it introduces concurrency-related complexities.

SELF-ASSESSMENT QUESTIONS – 1

1. What is multithreading in the context of Android app development?
   a) A programming technique that executes multiple apps simultaneously
   b) A way to execute multiple tasks in a single thread
   c) A programming technique that allows an app to execute multiple threads concurrently
   d) A way to execute tasks on a separate server
2. Why is multithreading important in Android app development?
   a) It reduces the number of threads, making the app simpler.
   b) It makes the user interface more complex.
   c) It ensures that long-running tasks do not make the UI unresponsive.
   d) It is necessary for downloading apps from the internet.
3. _____ is one of the benefits of multithreading in Android app development?
   a) Increased complexity of the code
   b) Sluggish and unresponsive user interface
   c) Improved performance by parallelizing tasks
   d) Reduced battery life

3. UNDERSTANDING THE ANDROID APPLICATION LIFECYCLE:

3.1 Overview of Android app lifecycle.

The Android application lifecycle defines the series of states and transitions that an Android app goes through from the moment it's launched until it's closed. Understanding the Android application lifecycle is crucial for proper resource management, user interface handling, and ensuring a seamless user experience. The Android application lifecycle includes the following states:

1. **Not Running (Initial State):** In this state, the app has not been started or has been terminated. It doesn't consume system resources, and no part of the app's code is running.
2. **Inactive State (onCreate()):** When a user launches the app, the system creates the app's process and invokes the `onCreate()` method in the app's main activity. This is where you typically initialize essential components, set up variables, and perform one-time setup.
3. **Running State (onStart() and onResume()):** After `onCreate()`, the app enters the Running state. In this state:
   `onStart()` is called, indicating that the app is visible but may not be in the foreground.
   `onResume()` is called when the app becomes fully visible and interactive. It's the point at which you can start retrieving data and updating the UI.
4. **Paused State (onPause()):** When the user interacts with another app or the device, the app can enter the Paused state. This is typically triggered by a foreground interruption. In `onPause()`, you might pause resource-intensive operations to ensure a responsive user experience.
5. **Stopped State (onStop()):** If the app is no longer visible (e.g., the user switches to another app or navigates to the home screen), `onStop()` is called. You might release resources that are not needed while the app is in the background.
6. **Terminated State (onDestroy()):** The app enters the Terminated state when the user closes it or the system needs to free up resources. In `onDestroy()`, you can perform final cleanup, such as releasing resources or saving user data.

The transitions between states:

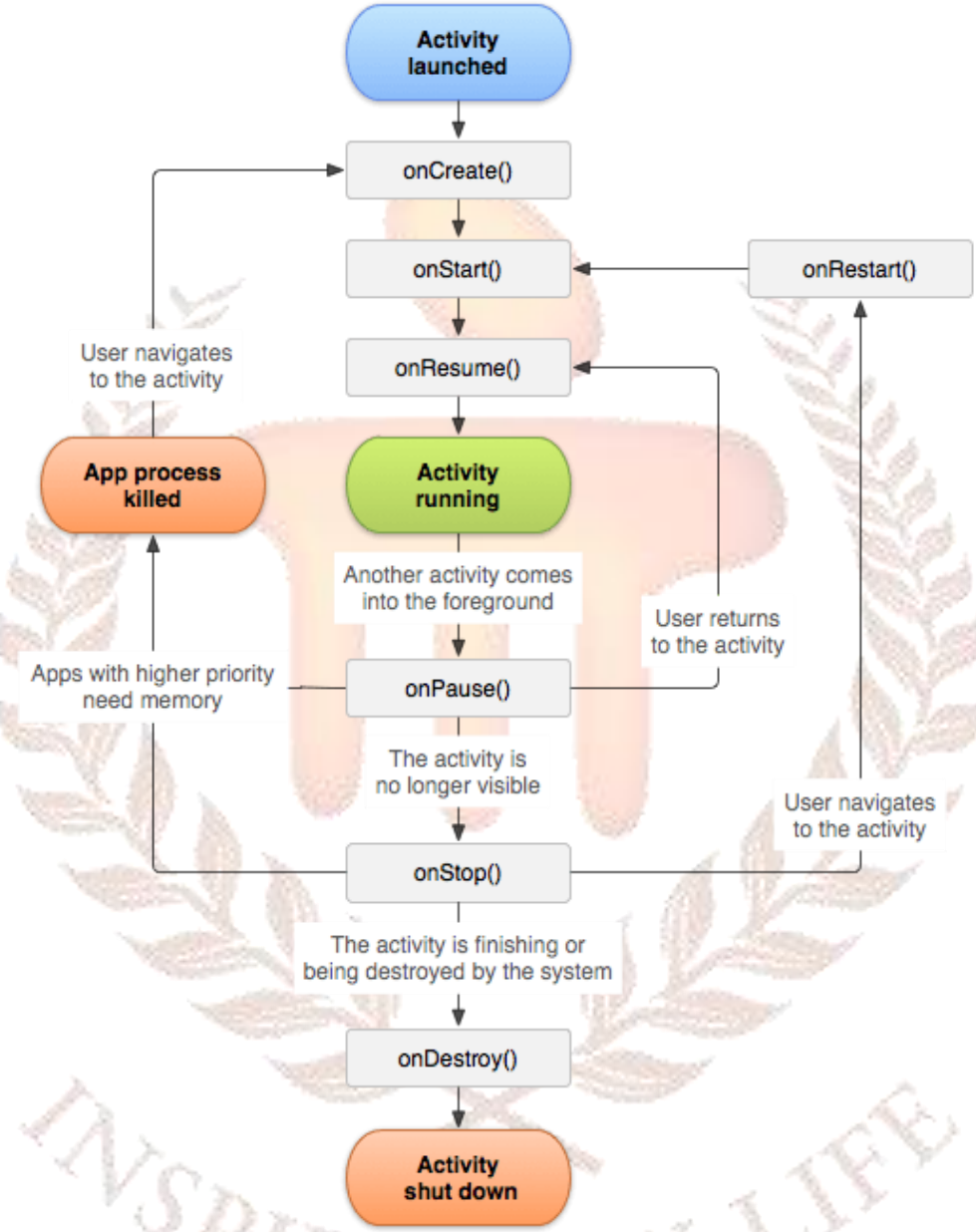
- App launch: `onCreate()` -> `onStart()` -> `onResume()`

- App goes to the background: `onPause()` -> `onStop()`

- App is resumed from the background: `onRestart()` -> `onStart()` -> `onResume()`

- App is closed by the user: `onPause()` -> `onStop()` -> `onDestroy()`

It's important to handle these lifecycle methods properly to ensure your app behaves as expected and to avoid common issues like memory leaks or unresponsive user interfaces. Additionally, Android provides tools like `ViewModels` and `SavedInstanceState` to help manage data and state across these lifecycle transitions, ensuring a seamless user experience.



(Image link : https://developer.android.com/guide/components/activities/activity-lifecycle)

Example:

Imagine you're developing a weather forecasting app. When a user launches your app, it goes through the following stages:

- **onCreate():** In this stage, you initialize essential components and variables.
- **onStart():** The app becomes visible, and any UI setup can happen here.
- **onResume():** The app is now in the foreground, and you can start retrieving weather data and updating the UI.
- **onPause():** If the user switches to another app, onPause() is called, allowing you to pause any resource-intensive tasks.
- **onStop():** If the app is no longer visible, onStop() is called, and you might release resources not needed in the background.
- **onDestroy():** When the app is closed, onDestroy() allows you to clean up any remaining resources.

3.2 Importance of the Main Thread in handling UI events.

The Main Thread, often referred to as the UI Thread, is the central thread responsible for managing the user interface and processing UI events. It's crucial to keep the Main Thread responsive because any delays or blocking operations on this thread can lead to a frozen or unresponsive app, resulting in a poor user experience.

The importance of the Main Thread in handling UI events:

1. **Responsiveness:** The Main Thread is responsible for rendering the user interface and processing user interactions. If this thread is blocked or delayed by resource-intensive tasks, the app becomes unresponsive, leading to a poor user experience. Users expect apps to respond promptly to their actions.
2. **UI Updates:** All UI updates, such as changing text, images, or layout, should be performed on the Main Thread. Any attempts to modify UI elements from background threads can result in crashes or unexpected behavior. This ensures consistency and avoids race conditions.
3. **User Interaction:** The Main Thread handles user interactions like button clicks, gestures, and touch events. These interactions are processed in real-time, and any delays can lead to input lag or dropped events, frustrating users.
4. **Animations and Transitions:** Smooth animations and transitions are essential for a polished user interface. These animations are typically driven by the Main Thread.

Blocking this thread can cause jittery animations or prevent them from running altogether.

5. **UI Responsiveness Metrics:** Both Android and iOS have built-in tools to monitor UI performance. They provide metrics like frame rate and response time. A slow Main Thread can result in low frame rates and sluggish interactions, which are easily detectable with these tools.
6. **Best Practices:** Following best practices, like offloading time-consuming tasks to background threads (e.g., using AsyncTask on Android or Grand Central Dispatch on iOS), helps maintain a responsive UI. This separation of tasks ensures that the UI Thread remains free to handle user input.
7. **User Satisfaction:** A responsive UI leads to improved user satisfaction and retention. Users are more likely to continue using an app that responds quickly to their actions and provides a seamless experience.

Example:

Suppose you're developing a messaging app. When a user taps the send button, the UI thread is responsible for updating the chat interface, showing a message "Sending..." and handling user interactions. Meanwhile, you need to send the message over the network in the background on a separate thread to avoid blocking the UI thread. If you don't do this, the app would freeze, and users won't be able to interact with it until the message is sent, leading to frustration.

3.3 How the Android OS manages threads.

Android OS manages threads to ensure the smooth operation of apps. It uses a combination of mechanisms like the Main Thread, AsyncTask, and thread pools to handle various tasks concurrently. Android provides tools and guidelines for developers to manage threads effectively.

Example:

Consider a photo-sharing app where users can upload images. When a user initiates an image upload, Android OS manages multiple threads:

- **The Main Thread** updates the UI to show the progress of the upload i.e., the Main Thread is responsible for handling the user interface and processing UI events, such as button clicks and touch gestures. It's crucial to keep the Main Thread responsive, as any delays or blocking operations on this thread can lead to an unresponsive app and a poor user experience.

  In your photo-sharing app, the Main Thread would update the UI to display progress, such as a progress bar or upload status, ensuring a smooth and interactive user interface.
- **A worker thread** (often managed by AsyncTask or a thread pool) handles the actual uploading of the image to the server. This ensures that the UI remains responsive, and the upload doesn't block the user from interacting with the app i.e.,
  - ✓ For tasks that are not related to UI operations, Android encourages developers to offload them to worker threads. This prevents long-running or resource-intensive tasks from blocking the Main Thread.
  - ✓ AsyncTask is a commonly used mechanism for simplifying the management of worker threads, allowing developers to execute background tasks and update the UI thread safely.
  - ✓ In your photo-sharing app, the actual image uploading process would be handled by a worker thread, ensuring that the UI thread remains free to handle user interactions.
- Thread Pools: If multiple uploads are in progress, a thread pool might manage these concurrently, controlling the number of simultaneous uploads to optimize resource usage i.e., In scenarios where multiple background tasks need to be executed concurrently, Android utilizes thread pools. Thread pools manage a set of worker threads and control the number of threads running simultaneously.

  This helps optimize resource usage and prevents excessive thread creation and destruction, which can be costly in terms of performance and resource consumption. In your app, if multiple users are uploading images simultaneously, a thread pool might be employed to efficiently manage and distribute these tasks among a limited number of threads.
- AsyncTask and Loaders (deprecated): AsyncTask was a popular choice for handling background tasks in earlier versions of Android. However, it's worth noting that

AsyncTask has been deprecated in recent Android versions, and alternatives like Kotlin Coroutines and RxJava are recommended for modern app development.

- Foreground Services and JobScheduler: For long-running tasks that need to execute in the background, Android provides tools like foreground services and the JobScheduler API to manage and schedule these tasks efficiently while considering factors like battery life and system resources.

SELF-ASSESSMENT QUESTIONS – 2

4. _____ is the Android application lifecycle?

a. A set of instructions for installing Android apps.

b. A sequence of states and transitions an Android app goes through.

c. The process of debugging Android apps.

d. A framework for designing Android app layouts.

5. In which state of the Android application lifecycle is the app not consuming system resources, and no part of its code is running?

a. Inactive State b. Running State

c. Stopped State d. Terminated State

6. Which method in Android's application lifecycle is typically used for initializing essential components and performing one-time setup?

a. onCreate()

b. onStart()

c. onResume()

7. What is the sequence of methods called when an Android app is resumed from the background?

a. onPause() -> onStop() -> onResume()

b. onRestart() -> onStart() -> onResume()

c. onCreate() -> onStart() -> onResume()

d. onPause() -> onResume() -> onStop()

8. Which method should be used for final cleanup and releasing resources when an Android app is closed by the user or terminated by the system?

a. onCreate()

b. onPause()

c. onStop()

d. onDestroy()

4. THE MAIN THREAD (UI THREAD):

The Main Thread, also known as the UI Thread, is the primary execution thread in a mobile application responsible for managing the user interface and processing UI events. It is a single-threaded environment, meaning that it can handle one task at a time sequentially. The Main Thread performs the following key functions:

- UI Rendering: The Main Thread is responsible for rendering the user interface elements such as buttons, text, images, and layout components on the screen. It updates the UI in response to user interactions and programmatic changes.
- User Interaction Processing: The Main Thread is the central hub for processing user interactions and events. It listens for user actions like taps, swipes, button clicks, and keyboard input.
- UI Event Handling: It processes user input and events like button clicks, touch gestures, and keyboard input. User interactions trigger callbacks and event handlers on the Main Thread.
- UI Updates: Any modifications to the UI, such as changing the text of a label, hiding or showing views, or animating UI elements, should be performed on the Main Thread to ensure thread safety and prevent conflicts.
- Synchronisation: The Main Thread ensures that access to UI elements and resources is synchronised, preventing multiple threads from simultaneously modifying UI components, which could lead to race conditions and unpredictable behavior.
- Responsiveness: Keeping the Main Thread responsive is crucial to providing a smooth and interactive user experience. Blocking or long-running operations on this thread can result in an unresponsive app, leading to a poor user experience.
- Thread Safety: The Main Thread enforces thread safety by ensuring that UI updates and modifications are performed sequentially. This prevents concurrent access and modification of UI elements, which can lead to unpredictable behavior and crashes.

4.1 Importance of keeping the Main Thread responsive:

Let us consider an example, a messaging app, when a user sends a message, the UI should update immediately to show the sent message. If the Main Thread is blocked, this update may be delayed, causing frustration for the user who expects instant feedback.

Here, Maintaining the Main Thread's responsiveness is indeed critical for delivering a positive user experience in mobile applications.

The importance of keeping the Main Thread responsive:

- User Expectations: Users expect modern mobile apps to respond promptly to their actions. When they tap a button, swipe, or interact with any UI element, they anticipate immediate feedback and visual changes.

In your messaging app example, when a user sends a message, they expect to see that message appear in the chat interface without delay. Any lag or unresponsiveness can lead to frustration and a perception of a poorly designed app.

- Perceived Speed: The responsiveness of the Main Thread directly affects the perceived speed of the app. Even if a task is processing in the background, if the Main Thread is responsive, the app can still feel fast and fluid to users. In contrast, if the Main Thread is blocked or delayed by background tasks, users might perceive the app as slow and unresponsive, even if those background tasks eventually complete successfully.
- User Engagement: A responsive Main Thread encourages user engagement. When the app responds quickly to user interactions, users are more likely to explore its features, interact with content, and spend more time within the app. Conversely, unresponsiveness can discourage users from using the app, resulting in lower engagement and retention rates.
- Consistency: Maintaining a responsive Main Thread ensures consistency in the user experience. When the UI behaves predictably and smoothly, users are more likely to trust the app and feel comfortable using it. Inconsistent or unpredictable UI behavior, caused by Main Thread delays, can erode user trust and confidence in the app.

- Preventing ANRs: Android's system monitors the Main Thread's responsiveness and may display an "Application Not Responding" (ANR) dialogue if it becomes blocked for an extended period. ANRs can lead to app crashes or force closures. By keeping the Main Thread responsive, developers can prevent ANRs and ensure their apps remain stable and reliable.
- Competitive Advantage: In a competitive app market, responsiveness can be a key differentiator. Apps that prioritise a responsive Main Thread tend to receive higher user ratings and reviews, ultimately leading to increased downloads and user retention.

4.2 Consequences Of Blocking The Main Thread.

Blocking the Main Thread can have serious consequences for your app. When the Main Thread is blocked for an extended period, the Android OS may perceive the app as unresponsive and display an "Application Not Responding" (ANR) dialog. ANRs are frustrating for users and can lead to app crashes or force closures.

Example: Consider a photo-sharing app where users can edit and apply filters to their photos. If the Main Thread becomes blocked while applying a filter, users might experience a delay in seeing the edited image, or worse, the app could trigger an ANR, causing the user to force close the app.

To avoid blocking the Main Thread and ensure a responsive UI, developers should offload time-consuming tasks to background threads, utilize techniques like AsyncTask or newer solutions like Kotlin Coroutines, and implement proper thread synchronisation when necessary. By doing so, you can maintain a smooth and enjoyable user experience in your Android app.

The significant consequences of blocking the Main Thread:

- ANRs (Application Not Responding): One of the most critical consequences of Main Thread blocking is the potential for ANRs. When the Main Thread is unresponsive for an extended period (usually around 5 seconds), the Android OS may trigger an ANR dialog, giving the user the option to wait or close the app forcibly.

ANRs are highly frustrating for users, and they can lead to a poor perception of the app's reliability and responsiveness.

- UI Freezing and Unresponsiveness: Blocking the Main Thread results in the app's UI becoming unresponsive during that time. Users may experience a frozen screen, delayed UI updates, and an inability to interact with the app. In your photo-sharing app example, if the Main Thread is blocked while applying a filter, users would perceive the app as slow and unresponsive during the editing process.
- Negative User Experience: A poor user experience can lead to decreased user satisfaction and retention. Users are more likely to abandon or uninstall an app that frequently exhibits unresponsiveness or ANRs. Apps that offer a smooth and responsive user interface tend to be more successful and receive better ratings and reviews.
- Interrupted Background Tasks: In addition to affecting the UI, Main Thread blocking can interfere with background tasks and services running in the app. For example, if a background task depends on the Main Thread for UI updates, it may also become blocked, impacting overall app functionality.
- Inconsistent Behavior: Blocking the Main Thread can result in inconsistent app behavior. Users might experience delays and unexpected behavior during their interactions with the app, leading to a lack of trust in its reliability.
- Reduced Competitiveness: In a competitive app market, apps that provide a responsive and seamless user experience tend to outperform and gain a competitive advantage over those with frequent Main Thread blocking issues.

SELF-ASSESSMENT QUESTIONS – 3

9. What is the primary execution thread responsible for managing the user interface and processing UI events in a mobile application called?
    a. Background Thread
    b. Worker Thread
    c. Main Thread (UI Thread)
    d. Secondary Thread
10. How can developers prevent ANRs (Application Not Responding) caused by Main Thread blocking?
    a. By displaying a warning message to users
    b. By implementing proper thread synchronisation
    c. By using larger thread stack sizes
    d. By increasing the app's memory allocation
11. Why is it essential to keep the Main Thread responsive in a mobile application?
    a. To save battery life
    b. To reduce memory usage
    c. To ensure efficient background processing
    d. To provide a smooth and interactive user experience
12. What happens when the Main Thread is blocked for an extended period in an Android app?
    a. It triggers a memory leak
    b. It displays an "Application Not Responding" (ANR) dialog
    c. It speeds up UI rendering
    d. It forces the app to close immediately

5. WORKER THREADS:

Worker threads, sometimes referred to as background threads, are threads other than the main UI thread that allow you to perform time-consuming tasks without affecting the user interface's responsiveness. In Android, they are crucial for executing operations such as network requests, database queries, or image processing without causing the app to become unresponsive.

5.1 Creating And Managing Worker Threads.

a. Creating worker Threads.

Creating a worker thread in Android can be accomplished in several ways, depending on the Android version and your preferred concurrency mechanism.

The below basic example shows creating a worker thread using Java threads:

Using Java Threads:

i. Create a Runnable:

First, create a class that implements the Runnable interface. This class will encapsulate the task you want to execute on the worker thread.

```
public class MyWorkerRunnable implements Runnable {
  @Override
  public void run() {
    // Your time-consuming task goes here
    // This code will execute on the worker thread
  }
}
```

ii. Create a Thread:

Next, create an instance of the Thread class and pass your Runnable as its constructor argument.

```
Thread workerThread = new Thread(new MyWorkerRunnable());
```

iii. Start the Thread:

To start the worker thread and execute the task, call the start() method on the thread instance.

```
workerThread.start();
```

iv. Wait for Completion (Optional):

If you need to wait for the worker thread to finish its task before proceeding with other operations, you can use the join() method.

```
try {
   workerThread.join(); // This will block until the worker thread completes
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

b. Managing Worker Threads

Managing worker threads effectively involves creating, controlling, and potentially terminating them when they're no longer needed.

```
// Start the worker thread
workerThread.start();


// Pause the worker thread (for example, after a user interaction)
if (workerThread.isAlive()) {
   try {
      workerThread.wait(); // Pauses the worker thread
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

```
        // Resume the worker thread (when needed)
        if (workerThread.isAlive()) {
            workerThread.notify(); // Resumes the worker thread
        }

        // Stop the worker thread (when it has completed its task)
        if (workerThread.isAlive()) {
            workerThread.interrupt(); // Stops the worker thread gracefully
        }
```

Communicating with the Worker Thread:

If you need to send data or instructions to the worker thread or receive results from it, you can use various mechanisms. Here's an example using a shared data structure:

```
public class MyWorkerRunnable implements Runnable {
    private String messageFromMainThread;

    public MyWorkerRunnable(String messageFromMainThread) {
        this.messageFromMainThread = messageFromMainThread;
    }

    @Override
    public void run() {
        // Access and use the messageFromMainThread
        // Perform time-consuming task

        // Communicate results back to the main thread
        String result = "Task completed";
        // You can use handlers, callbacks, or other mechanisms to send the result back to the main thread
    }
}
```

Handling Exceptions:

It's essential to handle exceptions that may occur during the execution of the worker thread:

```
public class MyWorkerRunnable implements Runnable {
   @Override
   public void run() {
      try {
         // Your time-consuming task goes here
      } catch (Exception e) {
         // Handle exceptions gracefully (e.g., log or notify the main thread)
         e.printStackTrace();
      }
   }
}
```

SELF-ASSESSMENT QUESTIONS – 4

13. What is the purpose of creating worker threads in Android?
    a. To handle all UI events
    b. To replace the Main Thread
    c. To execute time-consuming tasks without blocking the UI
    d. To simplify thread synchronization
14. In Android, how can you create a worker thread to perform background tasks?
    a. By directly instantiating a Thread object
    b. By subclassing the Main Thread
    c. By using AsyncTask exclusively
    d. By invoking a special method in the Activity class

6. THREAD MANAGEMENT IN ANDROID:

Thread management in Android involves creating and controlling threads efficiently to ensure smooth app performance. One common technique is to use a thread pool to manage and reuse threads, reducing the overhead of thread creation and destruction. Here's an example using

ThreadPoolExecutor.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadManagementExample {
   public static void main(String[] args) {
      // Create a thread pool with a fixed number of threads
      ExecutorService threadPool = Executors.newFixedThreadPool(3);
      // Submit tasks to the thread pool
      for (int i = 1; i <= 5; i++) {
         final int taskId = i;
         threadPool.execute(new Runnable() {
            public void run() {
               System.out.println("Task " + taskId + " is running on thread " + Thread.currentThread().getId());
            }
         });
      }

      // Shutdown the thread pool when done
      threadPool.shutdown();
   }
}
```

6.1 Thread synchronisation and coordination

In Android app development, multiple threads often need to coordinate their actions or access shared resources safely. The synchronised keyword can be used to create synchronised blocks that ensure only one thread can access the synchronised block at a time. Here's an example of thread synchronisation:

```
class SharedResource {
   private int count = 0;

   public synchronised void increment() {
      count++;
   }

   public synchronised int getCount() {
      return count;
   }
}

public class ThreadSyncExample {
   public static void main(String[] args) {
      SharedResource sharedResource = new SharedResource();

      Thread thread1 = new Thread(() -> {
         for (int i = 0; i < 10000; i++) {
            sharedResource.increment();
         }
      });

      Thread thread2 = new Thread(() -> {
         for (int i = 0; i < 10000; i++) {
            sharedResource.increment();
```

```
        }
    });

    thread1.start();
    thread2.start();

    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Final Count: " + sharedResource.getCount());
  }
}
```

6.2 Thread safety considerations

Thread safety is critical to prevent data corruption or unexpected behavior when multiple threads access shared resources concurrently. In Android, you should also consider thread safety when updating UI elements from background threads. Using runOnUiThread is a common way to ensure UI updates occur on the Main Thread.

```
import android.app.Activity;
public class ThreadSafetyExampleActivity extends Activity {
  private int count = 0;
  public void updateUI() {
    runOnUiThread(new Runnable() {
      public void run() {
        // Update UI elements here
        // e.g., textView.setText("Count: " + count);
      }
```

```
      });
   }
}
```


INSPIRED BY LIFE

7. ASYNCTASK IN ANDROID

AsyncTask is a class provided by the Android framework that simplifies the process of performing background tasks and updating the UI thread with the results. It is especially useful when you have a time-consuming operation that should not be executed on the main UI thread, such as downloading data from the internet or performing database operations.

The basic outline of an AsyncTask and its purpose:

- **AsyncTask** allows you to perform background operations (**doInBackground**()) without blocking the main UI thread.
- It provides a convenient way to report progress (**onProgressUpdate**()) and update the UI with the results of the background task (**onPostExecute**()).
- AsyncTask is typically used for tasks like network operations, file handling, or database operations.
- It helps maintain a responsive user interface by keeping long-running tasks off the main thread.

7.1 AsyncTask's three main methods: doInBackground(), onProgressUpdate(), and onPostExecute().

'AsyncTask' simplifies background task execution and UI updates by providing a structured way to perform these operations.

- 'doInBackground()': This method runs in a background thread and is where you place the time-consuming task you want to perform. For example, you can perform network requests, data processing, or any other operation that should not block the UI thread.

```
private class MyTask extends AsyncTask<Void, Void, String> {
    protected String doInBackground(Void... params) {
        // Background task
        String result = performBackgroundTask();
        return result;
    }
}
```

- onProgressUpdate(): This method runs on the UI thread and allows you to update the UI with progress information during the background task. You can use it to update progress bars or display intermediate results.

```
protected void onProgressUpdate(Integer... progress) {
   // Update UI with progress information
   progressBar.setProgress(progress[0]);
}
```

- onPostExecute(): This method runs on the UI thread and receives the result from doInBackground(). It's where you update the UI with the final result of the background task.

```
protected void onPostExecute(String result) {
   // Update UI with the result
   textView.setText(result);
}
```

By using AsyncTask, you ensure that UI updates are performed on the main thread (onProgressUpdate() and onPostExecute()), while the time-consuming task runs in the background (doInBackground()). This separation helps maintain a responsive user interface.

SELF-ASSESSMENT QUESTIONS – 5

15. What is the primary purpose of AsyncTask in Android development?
    a. To perform tasks on the main UI thread
    b. To simplify background task execution and UI updates
    c. To handle network connectivity
    d. To manage database operations
16. Which method in AsyncTask is responsible for performing time-consuming tasks in the background?
    a. onProgressUpdate()
    b. onPostExecute()
    c. doInBackground()
    d. doInBackgroundUI()

8. HANDLER AND LOOPER

Handler and Looper are fundamental classes in Android that play a crucial role in managing threads and facilitating communication between worker threads and the Main Thread (UI Thread).

- **Handler:** A Handler is an Android class that allows you to schedule tasks (messages and runnable) to be executed on a particular thread. It provides a way to send and process messages and is runnable asynchronously. Each Handler is associated with a Looper (explained below) and a specific thread.
- **Looper:** A Looper is responsible for running an event loop on a specific thread. It maintains a message queue continuously processes messages and is runnable in the order they are added to the queue. Android's Main Thread (UI Thread) has a default Looper, but additional Looper instances can be created for other threads.

8.1 Communication between worker threads and the Main Thread.

Handlers and Loopers are essential for facilitating communication between worker threads and the Main Thread:

- **Worker Threads:** In Android, it's common to create worker threads to perform background tasks like network requests or database operations. Worker threads can use Handlers to post messages or runnables to the Main Thread for UI updates or other actions. This is crucial because directly accessing UI elements from a worker thread is not allowed due to thread safety concerns.
- **Main Thread (UI Thread):** The Main Thread, responsible for the user interface, has a Looper associated with it by default. It continually processes messages and runnables posted by Handlers on its message queue. This enables worker threads to safely request UI updates without blocking the Main Thread.

The simplified example of how Handlers and Loopers facilitate communication:

```
// Create a Handler associated with the Main Thread's Looper
Handler mainThreadHandler = new Handler(Looper.getMainLooper());
// In a worker thread, post a Runnable to the Main Thread's message queue
```

```
workerThread.execute(() -> {
   // Perform background work
   // ...
   // Post a Runnable to update the UI on the Main Thread
   mainThreadHandler.post(() -> {
      // Update UI elements or perform UI-related tasks
   });
});
```

8.2 Implementing A Handler To Post Messages And Runnable.

To implement a Handler in your Android app and use it to post messages and runnables to the Main Thread, follow these steps:

1. **Create a Handler for the Main Thread:**

   You typically create a Handler associated with the Main Thread's Looper. This ensures that the messages and runnables posted to this Handler will be processed on the Main Thread.

   ```
   Handler mainThreadHandler = new Handler(Looper.getMainLooper());
   ```

2. **Post a Runnable from a Worker Thread:**

   In a worker thread (not the Main Thread), you can use the Handler to post a runnable for execution on the Main Thread. For example:

   ```
   workerThread.execute(() -> {
      // Perform background work
      // ...
      // Post a Runnable to update the UI on the Main Thread
      mainThreadHandler.post(() -> {
         // Update UI elements or perform UI-related tasks
   ```

```
    });
});
```

3. **Handle Messages and Runnables on the Main Thread:**

   On the Main Thread, you need to implement the logic to handle the messages and runnables posted by the worker threads. This is where you can update UI elements or perform any necessary actions.

   Here's a simplified example of how you might handle a posted runnable on the Main Thread:

```
mainThreadHandler.post(() -> {
    // This code runs on the Main Thread
    // Update UI elements or perform UI-related tasks
});
```

SELF-ASSESSMENT QUESTIONS – 6

17. What is the primary role of a Handler in Android?
    a. Running the main application logic
    b. Scheduling tasks to be executed on a specific thread
    c. Managing database operations
    d. Handling UI events
18. Which component in Android is responsible for running an event loop on a specific thread and maintaining a message queue?
    a. AsyncTask
    b. Handler
    c. Looper
    d. Runnable

9. SUMMARY

Multithreading is a fundamental concept in Android app development that allows applications to perform multiple tasks concurrently. It offers several benefits, including improved performance, responsiveness, and the ability to execute time-consuming operations in the background. However, multithreading also presents challenges related to synchronization, data sharing, and thread safety.

The Android application lifecycle governs how an app behaves from creation to termination. It's crucial for developers to understand this lifecycle to manage their app's components effectively. The Main Thread, responsible for handling UI events, plays a central role in maintaining a responsive user interface. Android OS manages threads to ensure smooth app operation and provides tools for developers to create responsive applications.

Keeping the Main Thread responsive is essential for delivering a positive user experience, as any delays or blocking operations can lead to an unresponsive app and user frustration. Blocking the Main Thread can result in Application Not Responding (ANR) errors, UI freezing, and inconsistent app behavior. Worker threads are utilized to execute time-consuming tasks in the background, preventing the Main Thread from becoming blocked and ensuring a seamless user experience.

Thread management in Android involves considerations such as synchronization and coordination to prevent data conflicts and race conditions. Thread safety is crucial to maintain the integrity of shared resources and prevent unexpected behavior. AsyncTask is a built-in Android class that simplifies background task execution by providing methods like doInBackground(), onProgressUpdate(), and onPostExecute(). Handlers and Loopers facilitate communication between worker threads and the Main Thread, enabling asynchronous message and runnable processing to update the UI and manage background tasks effectively.

10. ANSWERS

1. A programming technique that allows an app to execute multiple threads concurrently
2. It ensures that long-running tasks do not make the UI unresponsive
3. Improved performance by parallelizing tasks
4. A sequence of states and transitions an Android app goes through.
5. Stopped State
6. onCreate()
7. onRestart() -> onStart() -> onResume()
8. 8.onDestroy()
9. Main Thread (UI Thread)
10. By implementing proper thread Synchronisation
11. To provide a smooth and interactive user experience
12. It displays an "Application Not Responding" (ANR) dialog
13. To execute time-consuming tasks without blocking the UI
14. By directly instantiating a Thread object
15. To simplify background task execution and UI updates
16. doInBackground()
17. Scheduling tasks to be executed on a specific thread
18. Looper

11. TERMINAL QUESTIONS

1. Elucidate why Multithreading is Important in Android App Development.
2. Explicate the Android app life cycle.
3. Briefly explain the importance of the Main Thread in handling UI events
4. Describe how the android manages threads.
5. Write a short note on the Main Thread.
6. Write a code to create and manage the worker threads
7. Write a short note on AsyncTask in Android.
8. Briefly explain the importance of implementing a Handler to post messages and runnable.

Terminal Answers

1. Refer to section 1.1
2. Refer to section 2.1
3. Refer to section 2.2
4. Refer to section 2.3
5. Refer to section 3
6. Refer to section 4
7. Refer to section 6
8. Refer to section 7.2