# RLMCA234 Mobile Application Development Lab

July 2025

## Cycle 1

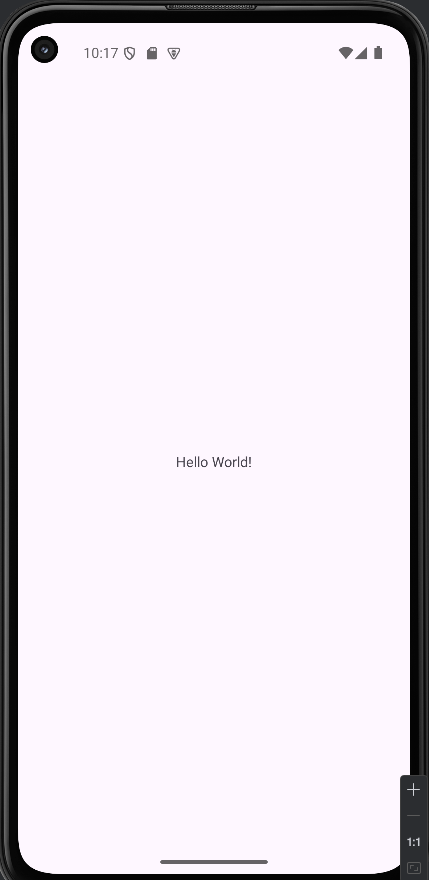
1. Execute the Hello World program on android studio.

Xml

<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:id="@+id/main"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MainActivity">  
  
 <TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Hello World!"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>

java

package com.example.helloworldbatch2;  
  
import android.os.Bundle;  
  
import androidx.activity.EdgeToEdge;  
import androidx.appcompat.app.AppCompatActivity;  
import androidx.core.graphics.Insets;  
import androidx.core.view.ViewCompat;  
import androidx.core.view.WindowInsetsCompat;  
  
public class MainActivity extends AppCompatActivity {  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 EdgeToEdge.*enable*(this);  
 setContentView(R.layout.*activity\_main*);  
 ViewCompat.*setOnApplyWindowInsetsListener*(findViewById(R.id.*main*), (v, insets) -> {  
 Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.*systemBars*());  
 v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);  
 return insets;  
 });  
 }  
}



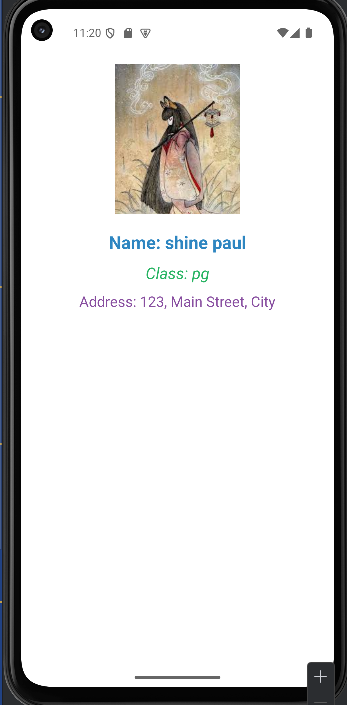
1. Develop an android mobile application to display your details – Name, Class , Address , Photo etc. (use of font colors, size, style…)

java

package com.example.helloworldbatch2;  
  
import android.os.Bundle;  
  
import androidx.activity.EdgeToEdge;  
import androidx.appcompat.app.AppCompatActivity;  
import androidx.core.graphics.Insets;  
import androidx.core.view.ViewCompat;  
import androidx.core.view.WindowInsetsCompat;  
  
public class MainActivity extends AppCompatActivity {  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 EdgeToEdge.*enable*(this);  
 setContentView(R.layout.*activity\_main*);  
 ViewCompat.*setOnApplyWindowInsetsListener*(findViewById(R.id.*main*), (v, insets) -> {  
 Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.*systemBars*());  
 v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);  
 return insets;  
 });  
 }  
}

xml

<?xml version="1.0" encoding="utf-8"?>  
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@+id/main"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:background="#FFFFFF">  
  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:orientation="vertical"  
 android:padding="20dp"  
 android:gravity="center\_horizontal">  
  
 <ImageView  
 android:id="@+id/profileImage"  
 android:layout\_width="157dp"  
 android:layout\_height="188dp"  
 android:layout\_marginBottom="20dp"  
 android:scaleType="centerCrop"  
 android:src="@drawable/shine" />  
  
 <TextView  
 android:id="@+id/nameText"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Name: shine paul"  
 android:textColor="#2E86C1"  
 android:textSize="22sp"  
 android:textStyle="bold" />  
  
 <TextView  
 android:id="@+id/classText"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="10dp"  
 android:text="Class: pg"  
 android:textColor="#28B463"  
 android:textSize="20sp"  
 android:textStyle="italic" />  
  
 <TextView  
 android:id="@+id/addressText"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="10dp"  
 android:text="Address: 123, Main Street, City"  
 android:textColor="#884EA0"  
 android:textSize="18sp" />  
  
 </LinearLayout>  
</ScrollView>



1. Develop an android mobile application to display some Question and answer in text boxes when a button event occurs. Set some questions in Text Views and display corresponding answers when a Button click event happens.

Que.java

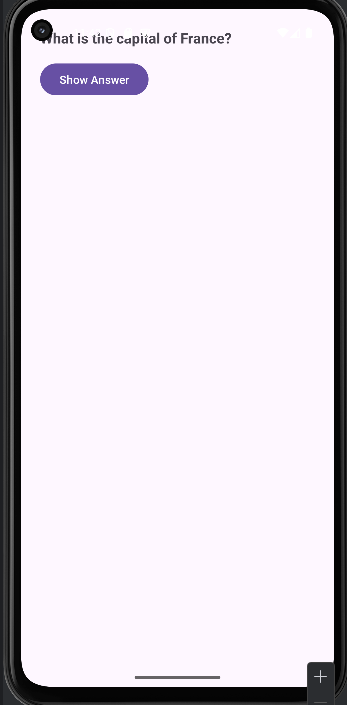
/\*  
 \* Copyright (C) 2006 The Android Open Source Project  
 \*  
 \* Licensed under the Apache License, Version 2.0 (the "License");  
 \* you may not use this file except in compliance with the License.  
 \* You may obtain a copy of the License at  
 \*  
 \* http://www.apache.org/licenses/LICENSE-2.0  
 \*  
 \* Unless required by applicable law or agreed to in writing, software  
 \* distributed under the License is distributed on an "AS IS" BASIS,  
 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 \* See the License for the specific language governing permissions and  
 \* limitations under the License.  
 \*/  
  
package android.os;  
  
import android.annotation.IntDef;  
import android.annotation.NonNull;  
import android.annotation.TestApi;  
import android.compat.annotation.UnsupportedAppUsage;  
import android.util.Log;  
import android.util.Printer;  
import android.util.SparseArray;  
import android.util.proto.ProtoOutputStream;  
  
import java.io.FileDescriptor;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.util.ArrayList;  
  
*/\*\**  
 *\* Low-level class holding the list of messages to be dispatched by a*  
 *\* {@link Looper}. Messages are not added directly to a MessageQueue,*  
 *\* but rather through {@link Handler} objects associated with the Looper.*  
 *\**  
 *\* <p>You can retrieve the MessageQueue for the current thread with*  
 *\* {@link Looper#myQueue() Looper.myQueue()}.*  
 *\*/*  
@android.ravenwood.annotation.RavenwoodKeepWholeClass  
@android.ravenwood.annotation.RavenwoodNativeSubstitutionClass(  
 "com.android.platform.test.ravenwood.nativesubstitution.MessageQueue\_host")  
public final class MessageQueue {  
 private static final String *TAG* = "MessageQueue";  
 private static final boolean *DEBUG* = false;  
  
 // True if the message queue can be quit.  
 @UnsupportedAppUsage  
 private final boolean mQuitAllowed;  
  
 @UnsupportedAppUsage  
 @SuppressWarnings("unused")  
 private long mPtr; // used by native code  
  
 @UnsupportedAppUsage  
 Message mMessages;  
 private Message mLast;  
 @UnsupportedAppUsage  
 private final ArrayList<IdleHandler> mIdleHandlers = new ArrayList<IdleHandler>();  
 private SparseArray<FileDescriptorRecord> mFileDescriptorRecords;  
 private IdleHandler[] mPendingIdleHandlers;  
 private boolean mQuitting;  
  
 // Indicates whether next() is blocked waiting in pollOnce() with a non-zero timeout.  
 private boolean mBlocked;  
  
 // Tracks the number of async message. We use this in enqueueMessage() to avoid searching the  
 // queue for async messages when inserting a message at the tail.  
 private int mAsyncMessageCount;  
  
 // The next barrier token.  
 // Barriers are indicated by messages with a null target whose arg1 field carries the token.  
 @UnsupportedAppUsage  
 private int mNextBarrierToken;  
  
 private native static long nativeInit();  
 private native static void nativeDestroy(long ptr);  
 @UnsupportedAppUsage  
 private native void nativePollOnce(long ptr, int timeoutMillis); /\*non-static for callbacks\*/  
 private native static void nativeWake(long ptr);  
 private native static boolean nativeIsPolling(long ptr);  
 private native static void nativeSetFileDescriptorEvents(long ptr, int fd, int events);  
  
 MessageQueue(boolean quitAllowed) {  
 mQuitAllowed = quitAllowed;  
 mPtr = *nativeInit*();  
 }  
  
 @Override  
 protected void finalize() throws Throwable {  
 try {  
 dispose();  
 } finally {  
 super.finalize();  
 }  
 }  
  
 // Disposes of the underlying message queue.  
 // Must only be called on the looper thread or the finalizer.  
 private void dispose() {  
 if (mPtr != 0) {  
 *nativeDestroy*(mPtr);  
 mPtr = 0;  
 }  
 }  
  
 */\*\**  
 *\* Returns true if the looper has no pending messages which are due to be processed.*  
 *\**  
 *\* <p>This method is safe to call from any thread.*  
 *\**  
 *\* @return True if the looper is idle.*  
 *\*/*  
public boolean isIdle() {  
 synchronized (this) {  
 final long now = SystemClock.*uptimeMillis*();  
 return mMessages == null || now < mMessages.when;  
 }  
 }  
  
 */\*\**  
 *\* Add a new {@link IdleHandler} to this message queue. This may be*  
 *\* removed automatically for you by returning false from*  
 *\* {@link IdleHandler#queueIdle IdleHandler.queueIdle()} when it is*  
 *\* invoked, or explicitly removing it with {@link #removeIdleHandler}.*  
 *\**  
 *\* <p>This method is safe to call from any thread.*  
 *\**  
 *\* @param handler The IdleHandler to be added.*  
 *\*/*  
public void addIdleHandler(@NonNull IdleHandler handler) {  
 if (handler == null) {  
 throw new NullPointerException("Can't add a null IdleHandler");  
 }  
 synchronized (this) {  
 mIdleHandlers.add(handler);  
 }  
 }  
  
 */\*\**  
 *\* Remove an {@link IdleHandler} from the queue that was previously added*  
 *\* with {@link #addIdleHandler}. If the given object is not currently*  
 *\* in the idle list, nothing is done.*  
 *\**  
 *\* <p>This method is safe to call from any thread.*  
 *\**  
 *\* @param handler The IdleHandler to be removed.*  
 *\*/*  
public void removeIdleHandler(@NonNull IdleHandler handler) {  
 synchronized (this) {  
 mIdleHandlers.remove(handler);  
 }  
 }  
  
 */\*\**  
 *\* Returns whether this looper's thread is currently polling for more work to do.*  
 *\* This is a good signal that the loop is still alive rather than being stuck*  
 *\* handling a callback. Note that this method is intrinsically racy, since the*  
 *\* state of the loop can change before you get the result back.*  
 *\**  
 *\* <p>This method is safe to call from any thread.*  
 *\**  
 *\* @return True if the looper is currently polling for events.*  
 *\* @hide*  
 *\*/*  
public boolean isPolling() {  
 synchronized (this) {  
 return isPollingLocked();  
 }  
 }  
  
 private boolean isPollingLocked() {  
 // If the loop is quitting then it must not be idling.  
 // We can assume mPtr != 0 when mQuitting is false.  
 return !mQuitting && *nativeIsPolling*(mPtr);  
 }  
  
 */\*\**  
 *\* Adds a file descriptor listener to receive notification when file descriptor*  
 *\* related events occur.*  
 *\* <p>*  
 *\* If the file descriptor has already been registered, the specified events*  
 *\* and listener will replace any that were previously associated with it.*  
 *\* It is not possible to set more than one listener per file descriptor.*  
 *\* </p><p>*  
 *\* It is important to always unregister the listener when the file descriptor*  
 *\* is no longer of use.*  
 *\* </p>*  
 *\**  
 *\* @param fd The file descriptor for which a listener will be registered.*  
 *\* @param events The set of events to receive: a combination of the*  
 *\* {@link OnFileDescriptorEventListener#EVENT\_INPUT},*  
 *\* {@link OnFileDescriptorEventListener#EVENT\_OUTPUT}, and*  
 *\* {@link OnFileDescriptorEventListener#EVENT\_ERROR} event masks. If the requested*  
 *\* set of events is zero, then the listener is unregistered.*  
 *\* @param listener The listener to invoke when file descriptor events occur.*  
 *\**  
 *\* @see OnFileDescriptorEventListener*  
 *\* @see #removeOnFileDescriptorEventListener*  
 *\*/*  
@android.ravenwood.annotation.RavenwoodThrow(blockedBy = android.os.ParcelFileDescriptor.class)  
 public void addOnFileDescriptorEventListener(@NonNull FileDescriptor fd,  
 @OnFileDescriptorEventListener.Events int events,  
 @NonNull OnFileDescriptorEventListener listener) {  
 if (fd == null) {  
 throw new IllegalArgumentException("fd must not be null");  
 }  
 if (listener == null) {  
 throw new IllegalArgumentException("listener must not be null");  
 }  
  
 synchronized (this) {  
 updateOnFileDescriptorEventListenerLocked(fd, events, listener);  
 }  
 }  
  
 */\*\**  
 *\* Removes a file descriptor listener.*  
 *\* <p>*  
 *\* This method does nothing if no listener has been registered for the*  
 *\* specified file descriptor.*  
 *\* </p>*  
 *\**  
 *\* @param fd The file descriptor whose listener will be unregistered.*  
 *\**  
 *\* @see OnFileDescriptorEventListener*  
 *\* @see #addOnFileDescriptorEventListener*  
 *\*/*  
@android.ravenwood.annotation.RavenwoodThrow(blockedBy = android.os.ParcelFileDescriptor.class)  
 public void removeOnFileDescriptorEventListener(@NonNull FileDescriptor fd) {  
 if (fd == null) {  
 throw new IllegalArgumentException("fd must not be null");  
 }  
  
 synchronized (this) {  
 updateOnFileDescriptorEventListenerLocked(fd, 0, null);  
 }  
 }  
  
 @android.ravenwood.annotation.RavenwoodThrow(blockedBy = android.os.ParcelFileDescriptor.class)  
 private void updateOnFileDescriptorEventListenerLocked(FileDescriptor fd, int events,  
 OnFileDescriptorEventListener listener) {  
 final int fdNum = fd.getInt$();  
  
 int index = -1;  
 FileDescriptorRecord record = null;  
 if (mFileDescriptorRecords != null) {  
 index = mFileDescriptorRecords.indexOfKey(fdNum);  
 if (index >= 0) {  
 record = mFileDescriptorRecords.valueAt(index);  
 if (record != null && record.mEvents == events) {  
 return;  
 }  
 }  
 }  
  
 if (events != 0) {  
 events |= OnFileDescriptorEventListener.*EVENT\_ERROR*;  
 if (record == null) {  
 if (mFileDescriptorRecords == null) {  
 mFileDescriptorRecords = new SparseArray<FileDescriptorRecord>();  
 }  
 record = new FileDescriptorRecord(fd, events, listener);  
 mFileDescriptorRecords.put(fdNum, record);  
 } else {  
 record.mListener = listener;  
 record.mEvents = events;  
 record.mSeq += 1;  
 }  
 *nativeSetFileDescriptorEvents*(mPtr, fdNum, events);  
 } else if (record != null) {  
 record.mEvents = 0;  
 mFileDescriptorRecords.removeAt(index);  
 *nativeSetFileDescriptorEvents*(mPtr, fdNum, 0);  
 }  
 }  
  
 // Called from native code.  
 @UnsupportedAppUsage(maxTargetSdk = Build.VERSION\_CODES.*R*, trackingBug = 170729553)  
 private int dispatchEvents(int fd, int events) {  
 // Get the file descriptor record and any state that might change.  
 final FileDescriptorRecord record;  
 final int oldWatchedEvents;  
 final OnFileDescriptorEventListener listener;  
 final int seq;  
 synchronized (this) {  
 record = mFileDescriptorRecords.get(fd);  
 if (record == null) {  
 return 0; // spurious, no listener registered  
 }  
  
 oldWatchedEvents = record.mEvents;  
 events &= oldWatchedEvents; // filter events based on current watched set  
 if (events == 0) {  
 return oldWatchedEvents; // spurious, watched events changed  
 }  
  
 listener = record.mListener;  
 seq = record.mSeq;  
 }  
  
 // Invoke the listener outside of the lock.  
 int newWatchedEvents = listener.onFileDescriptorEvents(  
 record.mDescriptor, events);  
 if (newWatchedEvents != 0) {  
 newWatchedEvents |= OnFileDescriptorEventListener.*EVENT\_ERROR*;  
 }  
  
 // Update the file descriptor record if the listener changed the set of  
 // events to watch and the listener itself hasn't been updated since.  
 if (newWatchedEvents != oldWatchedEvents) {  
 synchronized (this) {  
 int index = mFileDescriptorRecords.indexOfKey(fd);  
 if (index >= 0 && mFileDescriptorRecords.valueAt(index) == record  
 && record.mSeq == seq) {  
 record.mEvents = newWatchedEvents;  
 if (newWatchedEvents == 0) {  
 mFileDescriptorRecords.removeAt(index);  
 }  
 }  
 }  
 }  
  
 // Return the new set of events to watch for native code to take care of.  
 return newWatchedEvents;  
 }  
  
 @UnsupportedAppUsage  
 Message next() {  
 // Return here if the message loop has already quit and been disposed.  
 // This can happen if the application tries to restart a looper after quit  
 // which is not supported.  
 final long ptr = mPtr;  
 if (ptr == 0) {  
 return null;  
 }  
  
 int pendingIdleHandlerCount = -1; // -1 only during first iteration  
 int nextPollTimeoutMillis = 0;  
 for (;;) {  
 if (nextPollTimeoutMillis != 0) {  
 Binder.*flushPendingCommands*();  
 }  
  
 nativePollOnce(ptr, nextPollTimeoutMillis);  
  
 synchronized (this) {  
 // Try to retrieve the next message. Return if found.  
 final long now = SystemClock.*uptimeMillis*();  
 Message prevMsg = null;  
 Message msg = mMessages;  
 if (msg != null && msg.target == null) {  
 // Stalled by a barrier. Find the next asynchronous message in the queue.  
 do {  
 prevMsg = msg;  
 msg = msg.next;  
 } while (msg != null && !msg.isAsynchronous());  
 }  
 if (msg != null) {  
 if (now < msg.when) {  
 // Next message is not ready. Set a timeout to wake up when it is ready.  
 nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.*MAX\_VALUE*);  
 } else {  
 // Got a message.  
 mBlocked = false;  
 if (prevMsg != null) {  
 prevMsg.next = msg.next;  
 if (prevMsg.next == null) {  
 mLast = prevMsg;  
 }  
 } else {  
 mMessages = msg.next;  
 if (msg.next == null) {  
 mLast = null;  
 }  
 }  
 msg.next = null;  
 if (*DEBUG*) Log.*v*(*TAG*, "Returning message: " + msg);  
 msg.markInUse();  
 if (msg.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 return msg;  
 }  
 } else {  
 // No more messages.  
 nextPollTimeoutMillis = -1;  
 }  
  
 // Process the quit message now that all pending messages have been handled.  
 if (mQuitting) {  
 dispose();  
 return null;  
 }  
  
 // If first time idle, then get the number of idlers to run.  
 // Idle handles only run if the queue is empty or if the first message  
 // in the queue (possibly a barrier) is due to be handled in the future.  
 if (pendingIdleHandlerCount < 0  
 && (mMessages == null || now < mMessages.when)) {  
 pendingIdleHandlerCount = mIdleHandlers.size();  
 }  
 if (pendingIdleHandlerCount <= 0) {  
 // No idle handlers to run. Loop and wait some more.  
 mBlocked = true;  
 continue;  
 }  
  
 if (mPendingIdleHandlers == null) {  
 mPendingIdleHandlers = new IdleHandler[Math.*max*(pendingIdleHandlerCount, 4)];  
 }  
 mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);  
 }  
  
 // Run the idle handlers.  
 // We only ever reach this code block during the first iteration.  
 for (int i = 0; i < pendingIdleHandlerCount; i++) {  
 final IdleHandler idler = mPendingIdleHandlers[i];  
 mPendingIdleHandlers[i] = null; // release the reference to the handler  
  
 boolean keep = false;  
 try {  
 keep = idler.queueIdle();  
 } catch (Throwable t) {  
 Log.*wtf*(*TAG*, "IdleHandler threw exception", t);  
 }  
  
 if (!keep) {  
 synchronized (this) {  
 mIdleHandlers.remove(idler);  
 }  
 }  
 }  
  
 // Reset the idle handler count to 0 so we do not run them again.  
 pendingIdleHandlerCount = 0;  
  
 // While calling an idle handler, a new message could have been delivered  
 // so go back and look again for a pending message without waiting.  
 nextPollTimeoutMillis = 0;  
 }  
 }  
  
 void quit(boolean safe) {  
 if (!mQuitAllowed) {  
 throw new IllegalStateException("Main thread not allowed to quit.");  
 }  
  
 synchronized (this) {  
 if (mQuitting) {  
 return;  
 }  
 mQuitting = true;  
  
 if (safe) {  
 removeAllFutureMessagesLocked();  
 } else {  
 removeAllMessagesLocked();  
 }  
  
 // We can assume mPtr != 0 because mQuitting was previously false.  
 *nativeWake*(mPtr);  
 }  
 }  
  
 */\*\**  
 *\* Posts a synchronization barrier to the Looper's message queue.*  
 *\**  
 *\* Message processing occurs as usual until the message queue encounters the*  
 *\* synchronization barrier that has been posted. When the barrier is encountered,*  
 *\* later synchronous messages in the queue are stalled (prevented from being executed)*  
 *\* until the barrier is released by calling {@link #removeSyncBarrier} and specifying*  
 *\* the token that identifies the synchronization barrier.*  
 *\**  
 *\* This method is used to immediately postpone execution of all subsequently posted*  
 *\* synchronous messages until a condition is met that releases the barrier.*  
 *\* Asynchronous messages (see {@link Message#isAsynchronous} are exempt from the barrier*  
 *\* and continue to be processed as usual.*  
 *\**  
 *\* This call must be always matched by a call to {@link #removeSyncBarrier} with*  
 *\* the same token to ensure that the message queue resumes normal operation.*  
 *\* Otherwise the application will probably hang!*  
 *\**  
 *\* @return A token that uniquely identifies the barrier. This token must be*  
 *\* passed to {@link #removeSyncBarrier} to release the barrier.*  
 *\**  
 *\* @hide*  
 *\*/*  
@UnsupportedAppUsage  
 @TestApi  
 public int postSyncBarrier() {  
 return postSyncBarrier(SystemClock.*uptimeMillis*());  
 }  
  
 private int postSyncBarrier(long when) {  
 // Enqueue a new sync barrier token.  
 // We don't need to wake the queue because the purpose of a barrier is to stall it.  
 synchronized (this) {  
 final int token = mNextBarrierToken++;  
 final Message msg = Message.*obtain*();  
 msg.markInUse();  
 msg.when = when;  
 msg.arg1 = token;  
  
 if (Flags.messageQueueTailTracking() && mLast != null && mLast.when <= when) {  
 /\* Message goes to tail of list \*/  
 mLast.next = msg;  
 mLast = msg;  
 msg.next = null;  
 return token;  
 }  
  
 Message prev = null;  
 Message p = mMessages;  
 if (when != 0) {  
 while (p != null && p.when <= when) {  
 prev = p;  
 p = p.next;  
 }  
 }  
  
 if (p == null) {  
 /\* We reached the tail of the list, or list is empty. \*/  
 mLast = msg;  
 }  
  
 if (prev != null) { // invariant: p == prev.next  
 msg.next = p;  
 prev.next = msg;  
 } else {  
 msg.next = p;  
 mMessages = msg;  
 }  
 return token;  
 }  
 }  
  
 */\*\**  
 *\* Removes a synchronization barrier.*  
 *\**  
 *\* @param token The synchronization barrier token that was returned by*  
 *\* {@link #postSyncBarrier}.*  
 *\**  
 *\* @throws IllegalStateException if the barrier was not found.*  
 *\**  
 *\* @hide*  
 *\*/*  
@UnsupportedAppUsage  
 @TestApi  
 public void removeSyncBarrier(int token) {  
 // Remove a sync barrier token from the queue.  
 // If the queue is no longer stalled by a barrier then wake it.  
 synchronized (this) {  
 Message prev = null;  
 Message p = mMessages;  
 while (p != null && (p.target != null || p.arg1 != token)) {  
 prev = p;  
 p = p.next;  
 }  
 if (p == null) {  
 throw new IllegalStateException("The specified message queue synchronization "  
 + " barrier token has not been posted or has already been removed.");  
 }  
 final boolean needWake;  
 if (prev != null) {  
 prev.next = p.next;  
 if (prev.next == null) {  
 mLast = prev;  
 }  
 needWake = false;  
 } else {  
 mMessages = p.next;  
 if (mMessages == null) {  
 mLast = null;  
 }  
 needWake = mMessages == null || mMessages.target != null;  
 }  
 p.recycleUnchecked();  
  
 // If the loop is quitting then it is already awake.  
 // We can assume mPtr != 0 when mQuitting is false.  
 if (needWake && !mQuitting) {  
 *nativeWake*(mPtr);  
 }  
 }  
 }  
  
 boolean enqueueMessage(Message msg, long when) {  
 if (msg.target == null) {  
 throw new IllegalArgumentException("Message must have a target.");  
 }  
  
 synchronized (this) {  
 if (msg.isInUse()) {  
 throw new IllegalStateException(msg + " This message is already in use.");  
 }  
  
 if (mQuitting) {  
 IllegalStateException e = new IllegalStateException(  
 msg.target + " sending message to a Handler on a dead thread");  
 Log.*w*(*TAG*, e.getMessage(), e);  
 msg.recycle();  
 return false;  
 }  
  
 msg.markInUse();  
 msg.when = when;  
 Message p = mMessages;  
 boolean needWake;  
 if (p == null || when == 0 || when < p.when) {  
 // New head, wake up the event queue if blocked.  
 msg.next = p;  
 mMessages = msg;  
 needWake = mBlocked;  
 if (p == null) {  
 mLast = mMessages;  
 }  
 } else {  
 // Message is to be inserted at tail or middle of queue. Usually we don't have to  
 // wake up the event queue unless there is a barrier at the head of the queue and  
 // the message is the earliest asynchronous message in the queue.  
 needWake = mBlocked && p.target == null && msg.isAsynchronous();  
  
 // For readability, we split this portion of the function into two blocks based on  
 // whether tail tracking is enabled. This has a minor implication for the case  
 // where tail tracking is disabled. See the comment below.  
 if (Flags.messageQueueTailTracking()) {  
 if (when >= mLast.when) {  
 needWake = needWake && mAsyncMessageCount == 0;  
 msg.next = null;  
 mLast.next = msg;  
 mLast = msg;  
 } else {  
 // Inserted within the middle of the queue.  
 Message prev;  
 for (;;) {  
 prev = p;  
 p = p.next;  
 if (p == null || when < p.when) {  
 break;  
 }  
 if (needWake && p.isAsynchronous()) {  
 needWake = false;  
 }  
 }  
 if (p == null) {  
 /\* Inserting at tail of queue \*/  
 mLast = msg;  
 }  
 msg.next = p; // invariant: p == prev.next  
 prev.next = msg;  
 }  
 } else {  
 Message prev;  
 for (;;) {  
 prev = p;  
 p = p.next;  
 if (p == null || when < p.when) {  
 break;  
 }  
 if (needWake && p.isAsynchronous()) {  
 needWake = false;  
 }  
 }  
 msg.next = p; // invariant: p == prev.next  
 prev.next = msg;  
  
 /\*  
 \* If this block is executing then we have a build without tail tracking -  
 \* specifically: Flags.messageQueueTailTracking() == false. This is determined  
 \* at build time so the flag won't change on us during runtime.  
 \*  
 \* Since we don't want to pepper the code with extra checks, we only check  
 \* for tail tracking when we might use mLast. Otherwise, we continue to update  
 \* mLast as the tail of the list.  
 \*  
 \* In this case however we are not maintaining mLast correctly. Since we never  
 \* use it, this is fine. However, we run the risk of leaking a reference.  
 \* So set mLast to null in this case to avoid any Message leaks. The other  
 \* sites will never use the value so we are safe against null pointer derefs.  
 \*/  
 mLast = null;  
 }  
 }  
  
 if (msg.isAsynchronous()) {  
 mAsyncMessageCount++;  
 }  
  
 // We can assume mPtr != 0 because mQuitting is false.  
 if (needWake) {  
 *nativeWake*(mPtr);  
 }  
 }  
 return true;  
 }  
  
 boolean hasMessages(Handler h, int what, Object object) {  
 if (h == null) {  
 return false;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
 while (p != null) {  
 if (p.target == h && p.what == what && (object == null || p.obj == object)) {  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
 }  
 }  
  
 boolean hasEqualMessages(Handler h, int what, Object object) {  
 if (h == null) {  
 return false;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
 while (p != null) {  
 if (p.target == h && p.what == what && (object == null || object.equals(p.obj))) {  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
 }  
 }  
  
 @UnsupportedAppUsage(maxTargetSdk = Build.VERSION\_CODES.*R*, trackingBug = 170729553)  
 boolean hasMessages(Handler h, Runnable r, Object object) {  
 if (h == null) {  
 return false;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
 while (p != null) {  
 if (p.target == h && p.callback == r && (object == null || p.obj == object)) {  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
 }  
 }  
  
 boolean hasMessages(Handler h) {  
 if (h == null) {  
 return false;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
 while (p != null) {  
 if (p.target == h) {  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
 }  
 }  
  
 void removeMessages(Handler h, int what, Object object) {  
 if (h == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h && p.what == what  
 && (object == null || p.obj == object)) {  
 Message n = p.next;  
 mMessages = n;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 p = n;  
 }  
  
 if (p == null) {  
 mLast = mMessages;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && n.what == what  
 && (object == null || n.obj == object)) {  
 Message nn = n.next;  
 if (n.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 n.recycleUnchecked();  
 p.next = nn;  
 if (p.next == null) {  
 mLast = p;  
 }  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
 }  
  
 void removeEqualMessages(Handler h, int what, Object object) {  
 if (h == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h && p.what == what  
 && (object == null || object.equals(p.obj))) {  
 Message n = p.next;  
 mMessages = n;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 p = n;  
 }  
  
 if (p == null) {  
 mLast = mMessages;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && n.what == what  
 && (object == null || object.equals(n.obj))) {  
 Message nn = n.next;  
 if (n.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 n.recycleUnchecked();  
 p.next = nn;  
 if (p.next == null) {  
 mLast = p;  
 }  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
 }  
  
 void removeMessages(Handler h, Runnable r, Object object) {  
 if (h == null || r == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h && p.callback == r  
 && (object == null || p.obj == object)) {  
 Message n = p.next;  
 mMessages = n;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 p = n;  
 }  
  
 if (p == null) {  
 mLast = mMessages;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && n.callback == r  
 && (object == null || n.obj == object)) {  
 Message nn = n.next;  
 if (n.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 n.recycleUnchecked();  
 p.next = nn;  
 if (p.next == null) {  
 mLast = p;  
 }  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
 }  
  
 void removeEqualMessages(Handler h, Runnable r, Object object) {  
 if (h == null || r == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h && p.callback == r  
 && (object == null || object.equals(p.obj))) {  
 Message n = p.next;  
 mMessages = n;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 p = n;  
 }  
  
 if (p == null) {  
 mLast = mMessages;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && n.callback == r  
 && (object == null || object.equals(n.obj))) {  
 Message nn = n.next;  
 if (n.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 n.recycleUnchecked();  
 p.next = nn;  
 if (p.next == null) {  
 mLast = p;  
 }  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
 }  
  
  
 void removeCallbacksAndMessages(Handler h, Object object) {  
 if (h == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h  
 && (object == null || p.obj == object)) {  
 Message n = p.next;  
 mMessages = n;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 p = n;  
 }  
  
 if (p == null) {  
 mLast = mMessages;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && (object == null || n.obj == object)) {  
 Message nn = n.next;  
 if (n.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 n.recycleUnchecked();  
 p.next = nn;  
 if (p.next == null) {  
 mLast = p;  
 }  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
 }  
  
 void removeCallbacksAndEqualMessages(Handler h, Object object) {  
 if (h == null) {  
 return;  
 }  
  
 synchronized (this) {  
 Message p = mMessages;  
  
 // Remove all messages at front.  
 while (p != null && p.target == h  
 && (object == null || object.equals(p.obj))) {  
 Message n = p.next;  
 mMessages = n;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 p = n;  
 }  
  
 if (p == null) {  
 mLast = mMessages;  
 }  
  
 // Remove all messages after front.  
 while (p != null) {  
 Message n = p.next;  
 if (n != null) {  
 if (n.target == h && (object == null || object.equals(n.obj))) {  
 Message nn = n.next;  
 if (n.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 n.recycleUnchecked();  
 p.next = nn;  
 if (p.next == null) {  
 mLast = p;  
 }  
 continue;  
 }  
 }  
 p = n;  
 }  
 }  
 }  
  
 private void removeAllMessagesLocked() {  
 Message p = mMessages;  
 while (p != null) {  
 Message n = p.next;  
 p.recycleUnchecked();  
 p = n;  
 }  
 mMessages = null;  
 mLast = null;  
 mAsyncMessageCount = 0;  
 }  
  
 private void removeAllFutureMessagesLocked() {  
 final long now = SystemClock.*uptimeMillis*();  
 Message p = mMessages;  
 if (p != null) {  
 if (p.when > now) {  
 removeAllMessagesLocked();  
 } else {  
 Message n;  
 for (;;) {  
 n = p.next;  
 if (n == null) {  
 return;  
 }  
 if (n.when > now) {  
 break;  
 }  
 p = n;  
 }  
 p.next = null;  
 mLast = p;  
  
 do {  
 p = n;  
 n = p.next;  
 if (p.isAsynchronous()) {  
 mAsyncMessageCount--;  
 }  
 p.recycleUnchecked();  
 } while (n != null);  
 }  
 }  
 }  
  
 void dump(Printer pw, String prefix, Handler h) {  
 synchronized (this) {  
 long now = SystemClock.*uptimeMillis*();  
 int n = 0;  
 for (Message msg = mMessages; msg != null; msg = msg.next) {  
 if (h == null || h == msg.target) {  
 pw.println(prefix + "Message " + n + ": " + msg.toString(now));  
 }  
 n++;  
 }  
 pw.println(prefix + "(Total messages: " + n + ", polling=" + isPollingLocked()  
 + ", quitting=" + mQuitting + ")");  
 }  
 }  
  
 void dumpDebug(ProtoOutputStream proto, long fieldId) {  
 final long messageQueueToken = proto.start(fieldId);  
 synchronized (this) {  
 for (Message msg = mMessages; msg != null; msg = msg.next) {  
 msg.dumpDebug(proto, MessageQueueProto.MESSAGES);  
 }  
 proto.write(MessageQueueProto.IS\_POLLING\_LOCKED, isPollingLocked());  
 proto.write(MessageQueueProto.IS\_QUITTING, mQuitting);  
 }  
 proto.end(messageQueueToken);  
 }  
  
 */\*\**  
 *\* Callback interface for discovering when a thread is going to block*  
 *\* waiting for more messages.*  
 *\*/*  
public static interface IdleHandler {  
 */\*\**  
 *\* Called when the message queue has run out of messages and will now*  
 *\* wait for more. Return true to keep your idle handler active, false*  
 *\* to have it removed. This may be called if there are still messages*  
 *\* pending in the queue, but they are all scheduled to be dispatched*  
 *\* after the current time.*  
 *\*/*  
boolean queueIdle();  
 }  
  
 */\*\**  
 *\* A listener which is invoked when file descriptor related events occur.*  
 *\*/*  
public interface OnFileDescriptorEventListener {  
 */\*\**  
 *\* File descriptor event: Indicates that the file descriptor is ready for input*  
 *\* operations, such as reading.*  
 *\* <p>*  
 *\* The listener should read all available data from the file descriptor*  
 *\* then return <code>true</code> to keep the listener active or <code>false</code>*  
 *\* to remove the listener.*  
 *\* </p><p>*  
 *\* In the case of a socket, this event may be generated to indicate*  
 *\* that there is at least one incoming connection that the listener*  
 *\* should accept.*  
 *\* </p><p>*  
 *\* This event will only be generated if the {@link #EVENT\_INPUT} event mask was*  
 *\* specified when the listener was added.*  
 *\* </p>*  
 *\*/*  
public static final int *EVENT\_INPUT* = 1 << 0;  
  
 */\*\**  
 *\* File descriptor event: Indicates that the file descriptor is ready for output*  
 *\* operations, such as writing.*  
 *\* <p>*  
 *\* The listener should write as much data as it needs. If it could not*  
 *\* write everything at once, then it should return <code>true</code> to*  
 *\* keep the listener active. Otherwise, it should return <code>false</code>*  
 *\* to remove the listener then re-register it later when it needs to write*  
 *\* something else.*  
 *\* </p><p>*  
 *\* This event will only be generated if the {@link #EVENT\_OUTPUT} event mask was*  
 *\* specified when the listener was added.*  
 *\* </p>*  
 *\*/*  
public static final int *EVENT\_OUTPUT* = 1 << 1;  
  
 */\*\**  
 *\* File descriptor event: Indicates that the file descriptor encountered a*  
 *\* fatal error.*  
 *\* <p>*  
 *\* File descriptor errors can occur for various reasons. One common error*  
 *\* is when the remote peer of a socket or pipe closes its end of the connection.*  
 *\* </p><p>*  
 *\* This event may be generated at any time regardless of whether the*  
 *\* {@link #EVENT\_ERROR} event mask was specified when the listener was added.*  
 *\* </p>*  
 *\*/*  
public static final int *EVENT\_ERROR* = 1 << 2;  
  
 */\*\* @hide \*/*  
@Retention(RetentionPolicy.*SOURCE*)  
 @IntDef(flag = true, prefix = { "EVENT\_" }, value = {  
 *EVENT\_INPUT*,  
 *EVENT\_OUTPUT*,  
 *EVENT\_ERROR*  
})  
 public @interface Events {}  
  
 */\*\**  
 *\* Called when a file descriptor receives events.*  
 *\**  
 *\* @param fd The file descriptor.*  
 *\* @param events The set of events that occurred: a combination of the*  
 *\* {@link #EVENT\_INPUT}, {@link #EVENT\_OUTPUT}, and {@link #EVENT\_ERROR} event masks.*  
 *\* @return The new set of events to watch, or 0 to unregister the listener.*  
 *\**  
 *\* @see #EVENT\_INPUT*  
 *\* @see #EVENT\_OUTPUT*  
 *\* @see #EVENT\_ERROR*  
 *\*/*  
@Events int onFileDescriptorEvents(@NonNull FileDescriptor fd, @Events int events);  
 }  
  
 private static final class FileDescriptorRecord {  
 public final FileDescriptor mDescriptor;  
 public int mEvents;  
 public OnFileDescriptorEventListener mListener;  
 public int mSeq;  
  
 public FileDescriptorRecord(FileDescriptor descriptor,  
 int events, OnFileDescriptorEventListener listener) {  
 mDescriptor = descriptor;  
 mEvents = events;  
 mListener = listener;  
 }  
 }  
}

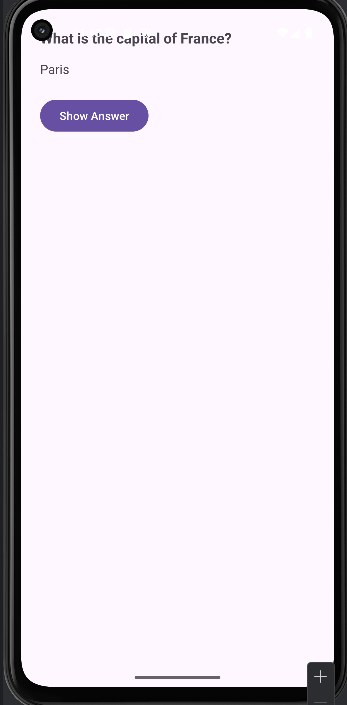
java

package com.example.helloworldbatch2;  
  
import android.os.Bundle;  
import android.view.View;  
import android.widget.Button;  
import android.widget.TextView;  
import androidx.appcompat.app.AppCompatActivity;  
  
public class MainActivity extends AppCompatActivity {  
  
 TextView textAnswer;  
 Button buttonShowAnswer;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*); // Make sure this file name is correct  
  
 // Get references  
 textAnswer = findViewById(R.id.*textAnswer*);  
 buttonShowAnswer = findViewById(R.id.*buttonShowAnswer*);  
  
 // Set click listener  
 buttonShowAnswer.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 textAnswer.setVisibility(View.*VISIBLE*);  
 }  
 });  
 }  
}

xml

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@+id/main"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"  
 android:padding="24dp">  
  
 <TextView  
 android:id="@+id/textQuestion"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="What is the capital of France?"  
 android:textSize="18sp"  
 android:textStyle="bold"  
 android:layout\_marginBottom="16dp" />  
  
 <TextView  
 android:id="@+id/textAnswer"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Paris"  
 android:textSize="16sp"  
 android:visibility="gone"  
 android:layout\_marginBottom="24dp" />  
  
 <Button  
 android:id="@+id/buttonShowAnswer"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Show Answer" />  
</LinearLayout>





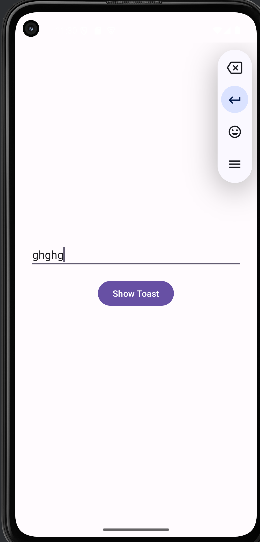
1. Develop an android mobile application which accept a string via editText and Toast it.

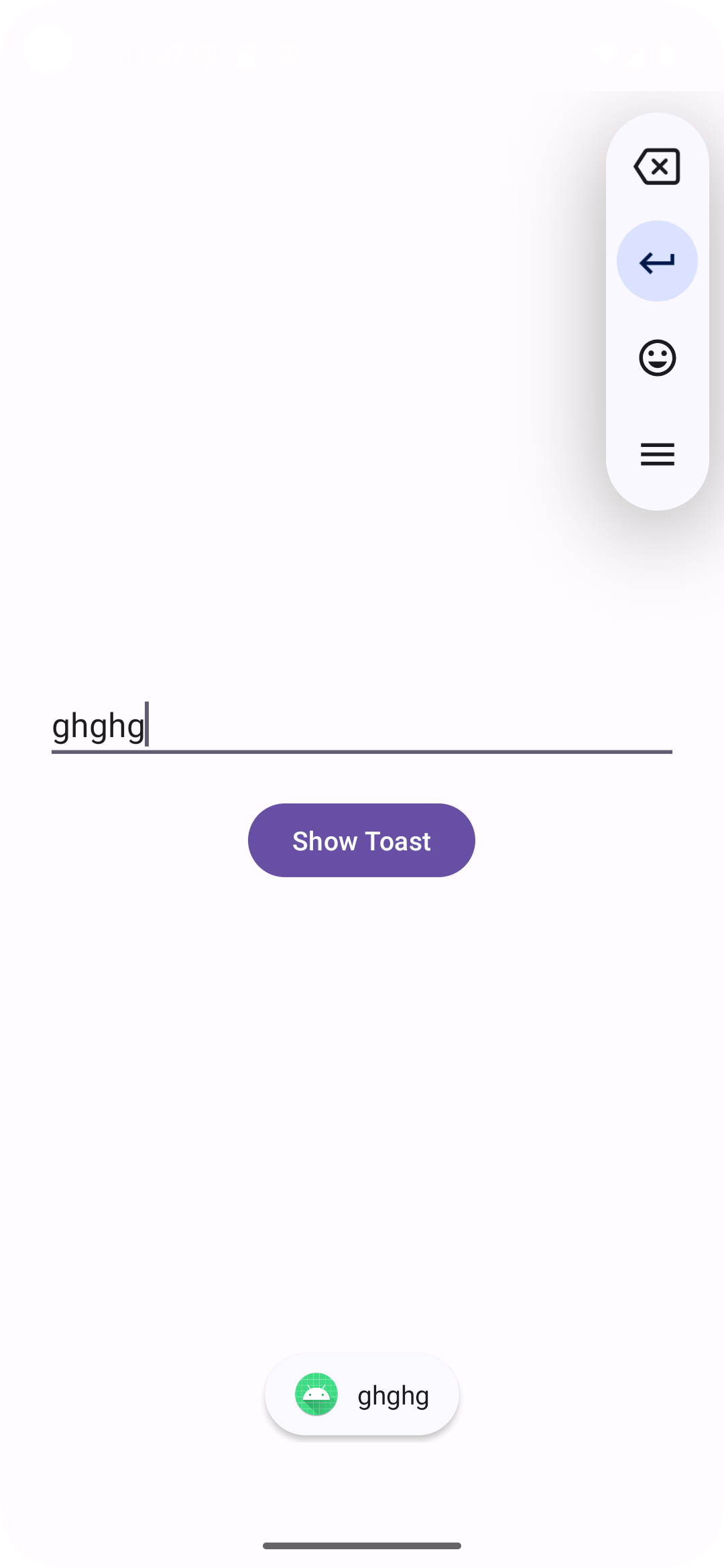
Xml

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"  
 android:padding="24dp"  
 android:gravity="center">  
  
 <EditText  
 android:id="@+id/editTextInput"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="Enter text here" />  
  
 <Button  
 android:id="@+id/buttonToast"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Show Toast"  
 android:layout\_marginTop="16dp"/>  
</LinearLayout>

java

package com.example.batch2q4;  
  
import android.os.Bundle;  
import android.widget.Toast;  
import android.widget.Button;  
import android.widget.EditText;  
import androidx.appcompat.app.AppCompatActivity;  
import com.example.batch2q4.R;  
  
  
public class MainActivity extends AppCompatActivity {  
  
 EditText editTextInput;  
 Button buttonToast;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*); // Ensure activity\_main.xml exists  
  
 editTextInput = findViewById(R.id.*editTextInput*);  
 buttonToast = findViewById(R.id.*buttonToast*);  
  
 buttonToast.setOnClickListener(v -> {  
 String input = editTextInput.getText().toString();  
 Toast.*makeText*(this, input, Toast.*LENGTH\_SHORT*).show();  
 });  
 }  
}





1. Develop an android mobile application to perform calculator functions. Accept inputs from GUI.