Table of Contents

1. Runtime Notification Permission

Category: Privacy and Security

Change Content: Android 13 requires all applications to explicitly request POST_NOTIFICATIONS permission to send notifications, even if the app targets a lower API level.

Change Date: August 2022

Reference Link: https://developer.android.com/develop/ui/views/notifications/notification-permission

Code Examples:

```
// Notification implementation for Android 10 and Android 13
private void showNotification() {
    // DIFFERENCE: Check and request notification permission in Android 13
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.POST_NOTIFICATIONS)
```

```
            != PackageManager.PERMISSION_GRANTED) {
        // Request notification permission
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.POST_NOTIFICATIONS},
                REQUEST_NOTIFICATION_PERMISSION);
        return; // Wait for permission grant before showing notification
    }
  }

  NotificationManager notificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

  // Create notification channel (required for Android 8.0+)
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    NotificationChannel channel = new NotificationChannel(
            "channel_id", "Channel Name",
            NotificationManager.IMPORTANCE_DEFAULT);
    notificationManager.createNotificationChannel(channel);
  }

  // Create and send notification
  NotificationCompat.Builder builder = new NotificationCompat.Builder(this, "channel_id")
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("Notification Title")
        .setContentText("Notification Content")
        .setPriority(NotificationCompat.PRIORITY_DEFAULT);

  notificationManager.notify(1, builder.build());
}

// Handle permission request result for Android 13
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
  super.onRequestPermissionsResult(requestCode, permissions, grantResults);
  if (requestCode == REQUEST_NOTIFICATION_PERMISSION) {
```

```
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // Permission granted, show notification
            showNotification();
        } else {
            // Permission denied, inform user or take alternative action
            Toast.makeText(this, "Notification permission needed to show notifications",
                    Toast.LENGTH_SHORT).show();
        }
    }
}
```

2. Foreground Service Type Declaration

Category: Application Lifecycle

Change Content: Android 13 requires all applications using foreground services to declare specific types in the manifest file, otherwise an exception will be thrown.

Change Date: August 2022

Reference Link: https://developer.android.com/guide/components/foreground-services

Code Examples:

```
<!-- AndroidManifest.xml for Android 10 and Android 13 -->
<service
    android:name=".MyForegroundService"
    android:enabled="true"
    android:exported="false"
    android:foregroundServiceType="location|camera|microphone" /> <!-- DIFFERENCE: Specify
foreground service type for Android 13 -->


// Foreground service implementation for Android 10 and Android 13
public class MyForegroundService extends Service {
    private static final int NOTIFICATION_ID = 1;
```

```
    @Override
    public void onCreate() {
        super.onCreate();

        createNotificationChannel();

        Notification notification = new NotificationCompat.Builder(this, "foreground_channel")
                .setContentTitle("Foreground Service")
                .setContentText("Service is running")
                .setSmallIcon(R.drawable.ic_notification)
                .build();

        // DIFFERENCE: Specify foreground service type for Android 10 and above
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            // Ensure declared foreground service type matches actual usage
            startForeground(NOTIFICATION_ID, notification,
ServiceInfo.FOREGROUND_SERVICE_TYPE_LOCATION);
        } else {
            startForeground(NOTIFICATION_ID, notification);
        }
    }

    // Other service methods...
}
```

3. Background Location Access Restrictions

Category: Privacy and Security

Change Content: Android 13 further restricts background location access, requiring more explicit permission declarations and user authorization flow.

Change Date: August 2022

Reference Link: https://developer.android.com/training/location/permissions

Code Examples:

```
// Requesting location permission for Android 10 and Android 13
private void requestLocationPermission() {
    // Step-by-step permission request
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED) {
        // Request foreground location permission
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                REQUEST_FOREGROUND_LOCATION);
    } else {
        // Have foreground permission, request background permission
        if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_BACKGROUND_LOCATION)
                != PackageManager.PERMISSION_GRANTED) {
            // DIFFERENCE: Show explanation dialog in Android 13
            new AlertDialog.Builder(this)
                    .setTitle("Background Location Permission Needed")
                    .setMessage("To provide location services when the app is not visible, we need background
location permission. "
                            + "Android 13 has stricter restrictions on this permission, please grant it in the next step.")
                    .setPositiveButton("Request Permission", (dialog, which) -> {
                        ActivityCompat.requestPermissions(this,
                                new String[]{Manifest.permission.ACCESS_BACKGROUND_LOCATION},
                                REQUEST_BACKGROUND_LOCATION);
                    })
                    .setNegativeButton("Cancel", (dialog, which) -> {
                        // User denied background permission, use foreground-only location
                        startForegroundOnlyLocationTracking();
                    })
                    .create()
                    .show();
        } else {
```

```
            // Have all necessary permissions, start location tracking
            startFullLocationTracking();
        }
    }
}

// Handle permission request results
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == REQUEST_FOREGROUND_LOCATION) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // After getting foreground location permission, request background permission
            requestBackgroundLocationIfNeeded();
        } else {
            Toast.makeText(this, "Location permission needed for this feature", Toast.LENGTH_SHORT).show();
        }
    } else if (requestCode == REQUEST_BACKGROUND_LOCATION) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // Got background location permission, start full location tracking
            startFullLocationTracking();
        } else {
            // User denied background location permission, use foreground only
            Toast.makeText(this, "App will only get location in foreground", Toast.LENGTH_SHORT).show();
            startForegroundOnlyLocationTracking();
        }
    }
}

// Start full location tracking (foreground and background)
private void startFullLocationTracking() {
    // Check if location services are enabled
    LocationManager locationManager = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
    boolean isLocationEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER) ||
```

```
        locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER);

    if (!isLocationEnabled) {
        // Location services not enabled, prompt user to enable
        new AlertDialog.Builder(this)
                .setTitle("Location Services Not Enabled")
                .setMessage("Please enable location services in settings")
                .setPositiveButton("Go to Settings", (dialog, which) -> {
                    Intent intent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
                    startActivity(intent);
                })
                .setNegativeButton("Cancel", null)
                .show();
        return;
    }

    fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);

    // DIFFERENCE: Use more battery-efficient location request config on Android 13
    LocationRequest locationRequest = LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_BALANCED_POWER_ACCURACY) // Use more power-
efficient accuracy
            .setInterval(30000) // Update every 30 seconds
            .setFastestInterval(15000) // Fastest every 15 seconds
            .setMaxWaitTime(60000); // Wait at most 60 seconds

    LocationCallback locationCallback = new LocationCallback() {
        @Override
        public void onLocationResult(LocationResult locationResult) {
            if (locationResult != null) {
                // Handle location updates
                for (Location location : locationResult.getLocations()) {
                    // Process location data, but be mindful of background processing
                    processLocationUpdateInBackground(location);
                }
            }
```

```
        }
    };

    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION)
            == PackageManager.PERMISSION_GRANTED) {
        fusedLocationClient.requestLocationUpdates(
                locationRequest, locationCallback, Looper.getMainLooper());
    }
}

// Start location tracking in foreground only
private void startForegroundOnlyLocationTracking() {
    // Similar to startFullLocationTracking, but stop location updates in onStop
    // ...

    // Add in onStop:
    @Override
    protected void onStop() {
        super.onStop();
        if (fusedLocationClient != null && locationCallback != null) {
            fusedLocationClient.removeLocationUpdates(locationCallback);
        }
    }

    // Resume in onStart:
    @Override
    protected void onStart() {
        super.onStart();
        if (hasLocationPermission()) {
            startForegroundOnlyLocationTracking();
        }
    }
}
```

4. Granular Media Permissions

Category: Privacy and Security

Change Content: Storage permissions are split into three separate permissions: READ_MEDIA_IMAGES, READ_MEDIA_VIDEO, and READ_MEDIA_AUDIO. Applications must request specific permissions.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/behavior-changes-13#granular-media-permissions

Code Examples:

```
// Request storage or media permissions for Android 10 and Android 13
private void requestMediaPermissions() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        // Android 13: Request specific media permissions
        boolean hasImagePermission = ContextCompat.checkSelfPermission(this,
                Manifest.permission.READ_MEDIA_IMAGES) == PackageManager.PERMISSION_GRANTED;
        boolean hasVideoPermission = ContextCompat.checkSelfPermission(this,
                Manifest.permission.READ_MEDIA_VIDEO) == PackageManager.PERMISSION_GRANTED;
        boolean hasAudioPermission = ContextCompat.checkSelfPermission(this,
                Manifest.permission.READ_MEDIA_AUDIO) == PackageManager.PERMISSION_GRANTED;

        List<String> permissionsToRequest = new ArrayList<>();

        if (!hasImagePermission) {
            permissionsToRequest.add(Manifest.permission.READ_MEDIA_IMAGES);
        }
        if (!hasVideoPermission) {
            permissionsToRequest.add(Manifest.permission.READ_MEDIA_VIDEO);
        }
        if (!hasAudioPermission) {
            permissionsToRequest.add(Manifest.permission.READ_MEDIA_AUDIO);
```

```
        }

        if (!permissionsToRequest.isEmpty()) {
            ActivityCompat.requestPermissions(this,
                    permissionsToRequest.toArray(new String[0]),
                    REQUEST_MEDIA_PERMISSIONS);
        } else {
            // Have all needed permissions
            loadMediaBasedOnPermissions();
        }
    } else {
        // Android 12 and below: Use READ_EXTERNAL_STORAGE
        requestStoragePermission();
    }
}

// Request storage permission for Android 10 and below
private void requestStoragePermission() {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_EXTERNAL_STORAGE)
            != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
                REQUEST_STORAGE_PERMISSION);
    } else {
        loadAllMedia();
    }
}

// Load media based on granted permissions
private void loadMediaBasedOnPermissions() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_MEDIA_IMAGES)
                == PackageManager.PERMISSION_GRANTED) {
            loadImages();
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_MEDIA_VIDEO)
```

```
            == PackageManager.PERMISSION_GRANTED) {
        loadVideos();
    }
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_MEDIA_AUDIO)
            == PackageManager.PERMISSION_GRANTED) {
        loadAudio();
    }
  } else {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_EXTERNAL_STORAGE)
            == PackageManager.PERMISSION_GRANTED) {
        loadAllMedia();
    }
  }
}

// Load all media for Android 10 and below
private void loadAllMedia() {
  ContentResolver resolver = getContentResolver();
  Cursor cursor = resolver.query(
        MediaStore.Files.getContentUri("external"),
        null,
        null,
        null,
        null);

  if (cursor != null) {
    while (cursor.moveToNext()) {
        // Process all media files...
    }
    cursor.close();
  }
}

// Load images
private void loadImages() {
  ContentResolver resolver = getContentResolver();
```

```
    Cursor cursor = resolver.query(
            MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
            null,
            null,
            null,
            null);

    if (cursor != null) {
        while (cursor.moveToNext()) {
            // Process images...
        }
        cursor.close();
    }
}

// Load videos
private void loadVideos() {
    ContentResolver resolver = getContentResolver();
    Cursor cursor = resolver.query(
            MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
            null,
            null,
            null,
            null);

    if (cursor != null) {
        while (cursor.moveToNext()) {
            // Process videos...
        }
        cursor.close();
    }
}

// Load audio
private void loadAudio() {
    ContentResolver resolver = getContentResolver();
```

```
    Cursor cursor = resolver.query(
            MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
            null,
            null,
            null,
            null);

    if (cursor != null) {
        while (cursor.moveToNext()) {
            // Process audio...
        }
        cursor.close();
    }
}
```

5. WiFi Permission Changes

Category: Connectivity

Change Content: NEARBY_WIFI_DEVICES permission replaces location permission requirements for certain WiFi-related functionality. Applications need to adapt to the new permission.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/behavior-changes-13#nearby-devices-permission

Code Examples:

```
// WiFi scanning implementation for Android 10 and Android 13
private void scanWifi() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        // Android 13: Check NEARBY_WIFI_DEVICES permission
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.NEARBY_WIFI_DEVICES)
                != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
```

```
                new String[]{Manifest.permission.NEARBY_WIFI_DEVICES},
                REQUEST_NEARBY_WIFI_DEVICES);
        return;
      }
    } else {
      // Android 12 and below: Check location permission
      if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION)
          != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                REQUEST_LOCATION_PERMISSION);
        return;
      }
    }

    WifiManager wifiManager = (WifiManager) getSystemService(Context.WIFI_SERVICE);
    if (!wifiManager.isWifiEnabled()) {
      Toast.makeText(this, "Please enable WiFi", Toast.LENGTH_SHORT).show();
      return;
    }

    wifiManager.startScan();
    List<ScanResult> scanResults = wifiManager.getScanResults();

    // Process scan results...
}


<!-- AndroidManifest.xml changes for Android 10 and Android 13 -->
<!-- Android 10: -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />

<!-- Android 13: -->
<uses-permission android:name="android.permission.NEARBY_WIFI_DEVICES"
```

```
        android:usesPermissionFlags="neverForLocation" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
```

6. Intent Filter Restrictions

Category: Inter-App Communication

Change Content: Android 13 enhances Intent Filter restrictions, requiring more explicit Intent filter declarations.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/behavior-changes-all

Code Examples:

```
<!-- AndroidManifest.xml for Android 10 and Android 13 -->
<!-- Android 10: No need to specify exported attribute -->
<activity android:name=".MyActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="https" android:host="example.com" />
  </intent-filter>
</activity>
<service android:name=".MyService">
  <intent-filter>
    <action android:name="com.example.app.ACTION_SERVICE" />
  </intent-filter>
</service>

<!-- Android 13: Must specify exported attribute -->
<activity
  android:name=".MyActivity"
  android:exported="true"> <!-- DIFFERENCE: Must explicitly declare exported attribute -->
```

```
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" /> <!-- DIFFERENCE: Add
BROWSABLE category -->
        <data android:scheme="https" android:host="example.com" />
    </intent-filter>
</activity>
<service
    android:name=".MyService"
    android:exported="false"> <!-- DIFFERENCE: Must explicitly declare exported attribute -->
    <intent-filter>
        <action android:name="com.example.app.ACTION_SERVICE" />
    </intent-filter>
</service>
```

7. Intent Receiver Export Declaration

Category: Inter-App Communication

Change Content: Android 13 has stricter requirements for the exported attribute of BroadcastReceivers, requiring explicit declaration.

Change Date: August 2022

Reference Link: https://developer.android.com/guide/components/broadcasts

Code Examples:

```
<!-- AndroidManifest.xml for Android 10 and Android 13 -->
<!-- Android 10: No need to specify exported attribute -->
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

```
<receiver android:name=".InternalReceiver">
    <intent-filter>
        <action android:name="com.example.app.INTERNAL_ACTION" />
    </intent-filter>
</receiver>

<!-- Android 13: Must specify exported attribute -->
<receiver
    android:name=".MyReceiver"
    android:exported="true"> <!-- DIFFERENCE: Must explicitly declare exported attribute -->
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
<receiver
    android:name=".InternalReceiver"
    android:exported="false"> <!-- DIFFERENCE: Must explicitly declare exported attribute -->
    <intent-filter>
        <action android:name="com.example.app.INTERNAL_ACTION" />
    </intent-filter>
</receiver>


// Registering receivers dynamically for Android 10 and Android 13
// Android 10: Register receiver without specifying export status
private void registerReceiverOld() {
    BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            // Handle broadcast
        }
    };

    IntentFilter filter = new IntentFilter("com.example.app.ACTION");
    registerReceiver(receiver, filter);
}
```

```
// Android 13: Register receiver with export status
private void registerReceiverNew() {
    BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            // Handle broadcast
        }
    };

    IntentFilter filter = new IntentFilter("com.example.app.ACTION");
    // DIFFERENCE: Specify export status
    registerReceiver(receiver, filter, Context.RECEIVER_NOT_EXPORTED);
}

// Sending broadcasts for Android 10 and Android 13
private void sendBroadcast() {
    // Send internal broadcast (target is receiver within own app)
    Intent internalIntent = new Intent("com.example.app.INTERNAL_ACTION");
    // DIFFERENCE: Specify target package name
    internalIntent.setPackage(getPackageName());
    sendBroadcast(internalIntent);

    // Send explicit broadcast (specify target component)
    Intent explicitIntent = new Intent(this, MyReceiver.class);
    explicitIntent.setAction("com.example.app.EXPLICIT_ACTION");
    sendBroadcast(explicitIntent);
}
```

8. Enhanced Background App Restrictions

Category: Application Lifecycle

Change Content: Android 13 has stricter requirements for background app restrictions, requiring more explicit declarations and user authorization flow.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/behavior-changes-13

Code Examples:

```
// Background service implementation for Android 10 and Android 13
public class BackgroundService extends Service {
    private static final int NOTIFICATION_ID = 1001;

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // Android 10: Directly starting an activity from the background
        Intent activityIntent = new Intent(this, TargetActivity.class);
        activityIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

        // DIFFERENCE: Direct start might work on Android 10, but not recommended
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.Q) {
            startActivity(activityIntent);
        } else {
            // Android 13: Use notification to start activity from background
            createNotificationChannel();

            // Create PendingIntent to start Activity
            PendingIntent pendingIntent = PendingIntent.getActivity(
                    this, 0, activityIntent, PendingIntent.FLAG_IMMUTABLE);

            // Create notification
            NotificationCompat.Builder builder = new NotificationCompat.Builder(this, "channel_id")
                    .setSmallIcon(R.drawable.ic_notification)
                    .setContentTitle("Attention Required")
                    .setContentText("Tap this notification to continue")
                    .setContentIntent(pendingIntent)
                    .setAutoCancel(true);

            // Show notification
```

```
        NotificationManager notificationManager =
                (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            // Check notification permission
            if (ActivityCompat.checkSelfPermission(this, Manifest.permission.POST_NOTIFICATIONS)
                    == PackageManager.PERMISSION_GRANTED) {
                notificationManager.notify(NOTIFICATION_ID, builder.build());
            }
        } else {
            notificationManager.notify(NOTIFICATION_ID, builder.build());
        }
    }

    return START_NOT_STICKY;
  }

  private void createNotificationChannel() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationChannel channel = new NotificationChannel(
                "channel_id",
                "Channel Name",
                NotificationManager.IMPORTANCE_DEFAULT);

        NotificationManager notificationManager = getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
  }

  // Other service methods...
}
```

9. Clipboard Access Restrictions

Category: Privacy and Security

Change Content: Android 13 restricts apps' ability to access clipboard content in the background and shows clipboard access notifications to users.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/features/copy-paste

Code Examples:

```
// Clipboard access implementation for Android 10 and Android 13
private void accessClipboard() {
    ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);

    // DIFFERENCE: Only read clipboard when app is in foreground for Android 13
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        if (isAppInForeground()) {
            if (clipboard.hasPrimaryClip()) {
                ClipData clipData = clipboard.getPrimaryClip();
                if (clipData != null && clipData.getItemCount() > 0) {
                    CharSequence text = clipData.getItemAt(0).getText();
                    if (text != null) {
                        // Use clipboard content
                        Log.d("Clipboard", "Clipboard content: " + text);
                    }
                }
            }
        }
    } else {
        // Android 10: Can read clipboard in foreground or background
        if (clipboard.hasPrimaryClip()) {
            ClipData clipData = clipboard.getPrimaryClip();
            if (clipData != null && clipData.getItemCount() > 0) {
```

```
            CharSequence text = clipData.getItemAt(0).getText();
            if (text != null) {
                // Use clipboard content
                Log.d("Clipboard", "Clipboard content: " + text);
            }
        }
    }
}

// Monitor clipboard changes
clipboard.addPrimaryClipChangedListener(new ClipboardManager.OnPrimaryClipChangedListener() {
    @Override
    public void onPrimaryClipChanged() {
        // DIFFERENCE: Only read when app is in foreground for Android 13
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            if (isAppInForeground()) {
                if (clipboard.hasPrimaryClip()) {
                    ClipData clipData = clipboard.getPrimaryClip();
                    if (clipData != null && clipData.getItemCount() > 0) {
                        // Process new clipboard content
                    }
                }
            }
        } else {
            // Android 10: Can read clipboard changes in foreground or background
            if (clipboard.hasPrimaryClip()) {
                ClipData clipData = clipboard.getPrimaryClip();
                if (clipData != null && clipData.getItemCount() > 0) {
                    // Process new clipboard content
                }
            }
        }
    }
});

// Mark sensitive data when setting clipboard content
```

```
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        ClipData clip = ClipData.newPlainText("Sensitive Data", "1234-5678-9012-3456");
        PersistableBundle extras = new PersistableBundle();
        extras.putBoolean(ClipDescription.EXTRA_IS_SENSITIVE, true);
        clip.getDescription().setExtras(extras);
        clipboard.setPrimaryClip(clip);
    } else {
        // No sensitive data marking in older versions
        ClipData clip = ClipData.newPlainText("Data", "1234-5678-9012-3456");
        clipboard.setPrimaryClip(clip);
    }
}

// Check if app is in foreground
private boolean isAppInForeground() {
    ActivityManager activityManager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    List<ActivityManager.RunningAppProcessInfo> appProcesses =
activityManager.getRunningAppProcesses();
    if (appProcesses == null) {
        return false;
    }

    final String packageName = getPackageName();
    for (ActivityManager.RunningAppProcessInfo appProcess : appProcesses) {
        if (appProcess.importance ==
ActivityManager.RunningAppProcessInfo.IMPORTANCE_FOREGROUND
                && appProcess.processName.equals(packageName)) {
            return true;
        }
    }
    return false;
}
```

10. Exact Alarm Permission

Category: System Functionality

Change Content: Android 13 requires apps to declare SCHEDULE_EXACT_ALARM permission to set exact alarms, otherwise only inexact alarms can be set.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/behavior-changes-13#alarms-api-changes

Code Examples:

```
<!-- AndroidManifest.xml for Android 13 -->
<uses-permission android:name="android.permission.SCHEDULE_EXACT_ALARM" />


// Schedule exact alarm for Android 10 and Android 13
private void scheduleExactAlarm() {
    AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);

    // DIFFERENCE: Check permission for exact alarms in Android 13
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        if (!alarmManager.canScheduleExactAlarms()) {
            // No permission, guide user to grant permission
            Intent intent = new Intent(Settings.ACTION_REQUEST_SCHEDULE_EXACT_ALARM);
            intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            intent.setData(Uri.parse("package:" + getPackageName()));
            startActivity(intent);
            return;
        }
    }

    Intent intent = new Intent(this, AlarmReceiver.class);
    PendingIntent pendingIntent = PendingIntent.getBroadcast(
            this, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT | PendingIntent.FLAG_IMMUTABLE);
```

```
    // Set to execute every hour
    long intervalMillis = 60 * 60 * 1000; // 1 hour
    long triggerTime = System.currentTimeMillis() + intervalMillis;

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        alarmManager.setExactAndAllowWhileIdle(
                AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent);
    } else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
        alarmManager.setExact(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent);
    } else {
        alarmManager.set(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent);
    }
}

// Schedule infrequent tasks for Android 10 and Android 13
private void scheduleInfrequentTasks() {
    // Use JobScheduler instead of AlarmManager
    JobScheduler jobScheduler = (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);

    JobInfo.Builder builder = new JobInfo.Builder(JOB_ID,
            new ComponentName(this, MyJobService.class))
            .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
            .setPeriodic(3 * 60 * 60 * 1000) // Execute every 3 hours
            .setRequiresDeviceIdle(true)
            .setRequiresBatteryNotLow(true);

    jobScheduler.schedule(builder.build());
}
```

11. App Links Verification Changes

Category: Inter-App Communication

Change Content: Android 13 changes the app links verification mechanism, requiring a stricter verification process and more explicit proof of domain ownership.

Change Date: August 2022

Reference Link: https://developer.android.com/training/app-links

Code Examples:

```
<!-- AndroidManifest.xml for Android 10 and Android 13 -->
<activity
    android:name=".DeepLinkActivity"
    android:exported="true"> <!-- DIFFERENCE: Must explicitly declare exported attribute for Android 13 --
>
    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="https" android:host="example.com" />
    </intent-filter>
</activity>


// Deep link handling for Android 10 and Android 13
public class DeepLinkActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_deep_link);

        // DIFFERENCE: Check app links status in Android 13
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
            checkAppLinkStatus();
        }

        // Handle deep link
        handleIntent(getIntent());
```

```
}

@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    handleIntent(intent);
}

private void handleIntent(Intent intent) {
    String action = intent.getAction();
    Uri data = intent.getData();

    if (Intent.ACTION_VIEW.equals(action) && data != null) {
        String path = data.getPath();
        String query = data.getQuery();

        // Handle link...
        Log.d("DeepLink", "Handling link: " + data.toString());
    }
}

// Android 13: Check app link verification status
private void checkAppLinkStatus() {
    try {
        DomainVerificationManager manager = getSystemService(DomainVerificationManager.class);
        if (manager != null) {
            DomainVerificationUserState userState =
                    manager.getDomainVerificationUserState(getPackageName());

            if (userState != null) {
                Map<String, Integer> hostToStateMap = userState.getHostToStateMap();
                boolean needUserVerification = false;

                for (String domain : hostToStateMap.keySet()) {
                    int state = hostToStateMap.get(domain);
                    if (state != DomainVerificationUserState.DOMAIN_STATE_VERIFIED) {
```

```
                    needUserVerification = true;
                    Log.d("AppLinks", domain + " needs user verification");
                }
            }

            if (needUserVerification) {
                // Guide user to manually enable app links
                promptUserToEnableAppLinks();
            }
        }
    } catch (PackageManager.NameNotFoundException e) {
        Log.e("AppLinks", "Failed to get app links status", e);
    }
}

private void promptUserToEnableAppLinks() {
    new AlertDialog.Builder(this)
        .setTitle("Enable App Links")
        .setMessage("Please enable app links in settings to open related URLs directly in the app")
        .setPositiveButton("Go to Settings", (dialog, which) -> {
            Intent intent = new Intent(Settings.ACTION_APP_OPEN_BY_DEFAULT_SETTINGS);
            Uri uri = Uri.parse("package:" + getPackageName());
            intent.setData(uri);
            startActivity(intent);
        })
        .setNegativeButton("Later", null)
        .show();
}
}
```

12. Language Preferences API Changes

Category: Internationalization

Change Content: Android 13 introduces app-level language preferences, allowing users to set language per app. Applications need to adapt to the new API.

Change Date: August 2022

Reference Link: https://developer.android.com/about/versions/13/features/app-languages

Code Examples:

```
// Set language in app
private void setAppLanguage(String languageCode) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        // Android 13: Use new app-level language API
        LocaleManager localeManager = getSystemService(LocaleManager.class);
        if (localeManager != null) {
            if (TextUtils.isEmpty(languageCode)) {
                // Use system default language
                localeManager.setApplicationLocales(LocaleList.getEmptyLocaleList());
            } else {
                // Set app-specific language
                localeManager.setApplicationLocales(
                        new LocaleList(Locale.forLanguageTag(languageCode)));
            }
        }
    } else {
        // Android 10: Use traditional way for older versions
        Locale locale = TextUtils.isEmpty(languageCode) ?
                Resources.getSystem().getConfiguration().getLocales().get(0) :
                new Locale(languageCode);

        Locale.setDefault(locale);

        Resources resources = getResources();
        Configuration config = new Configuration(resources.getConfiguration());
        config.setLocale(locale); // Android 10: config.locale = locale;
        resources.updateConfiguration(config, resources.getDisplayMetrics());
```

```
        // Save language setting
        SharedPreferences preferences = getSharedPreferences("settings", MODE_PRIVATE);
        preferences.edit().putString("language", languageCode).apply();

        // Recreate Activity to apply changes
        recreate();
    }
}

// Restore saved language setting in Application class
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();

        // Restore saved language setting
        SharedPreferences preferences = getSharedPreferences("settings", MODE_PRIVATE);
        String savedLanguage = preferences.getString("language", "");

        if (!TextUtils.isEmpty(savedLanguage)) {
            Locale locale = new Locale(savedLanguage);
            Locale.setDefault(locale);

            Resources resources = getResources();
            Configuration config = resources.getConfiguration();
            config.setLocale(locale); // Android 10: config.locale = locale;
            resources.updateConfiguration(config, resources.getDisplayMetrics());
        }
    }
}

// Provide language selection UI
private void showLanguageSelectionDialog() {
    final String[] languages = {"System Default", "English", "Simplified Chinese", "Español", "日本語"};
    final String[] languageCodes = {"", "en", "zh", "es", "ja"};
```

```
    new AlertDialog.Builder(this)
            .setTitle("Select App Language")
            .setItems(languages, (dialog, which) -> {
                setAppLanguage(languageCodes[which]);
                if (Build.VERSION.SDK_INT < Build.VERSION_CODES.TIRAMISU) {
                    // Android 12 and below need Activity recreation
                    recreate();
                }
                // Android 13 applies language changes automatically
            })
            .show();
}
```

13. App Hibernation Improvements

Category: System Performance

Change Content: Android 13 enhances the app hibernation mechanism, managing unused apps more intelligently to reduce system resource usage.

Change Date: August 2022

Reference Link: https://developer.android.com/topic/performance/app-hibernation

Code Examples:

```
// Check battery optimization or app hibernation status
private void checkAppStatus() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        // Android 13: Check app hibernation status
        AppHibernationManager hibernationManager = getSystemService(AppHibernationManager.class);

        if (hibernationManager != null) {
            boolean isHibernatingForUser = hibernationManager.isHibernatingForUser(getPackageName());

            if (isHibernatingForUser) {
```

```
            Log.d("Hibernation", "App is in user hibernation state");
            // Detected hibernation state, reduce resource usage accordingly
            adaptToHibernation();
        }
    }
} else {
    // Android 10: Check battery optimization
    PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
    String packageName = getPackageName();

    boolean isIgnoringBatteryOptimizations =
            powerManager.isIgnoringBatteryOptimizations(packageName);

    if (!isIgnoringBatteryOptimizations) {
        // Request to ignore battery optimization
        Intent intent = new Intent(Settings.ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS);
        intent.setData(Uri.parse("package:" + packageName));
        startActivity(intent);
    }
}
}

// Adaptation measures for hibernation state
private void adaptToHibernation() {
    // Cancel non-essential background work
    WorkManager workManager = WorkManager.getInstance(this);
    workManager.cancelAllWorkByTag("non_essential");

    // Reduce cache size
    clearNonEssentialCache();

    // Reduce refresh frequency
    reduceSyncFrequency();
}

// When app resumes from hibernation
```

```
private void onAppResumeFromHibernation() {
    // Reinitialize necessary components
    refreshData();

    // Resume normal work scheduling
    scheduleNormalBackgroundWork();
}

// Schedule background work based on app status
private void scheduleBackgroundWork() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        // Android 13: Design progressive background work strategy
        scheduleAdaptiveBackgroundWork();
    } else {
        // Android 10: Schedule periodic background work
        WorkManager workManager = WorkManager.getInstance(this);

        PeriodicWorkRequest workRequest = new PeriodicWorkRequest.Builder(
                SyncWorker.class,
                1, TimeUnit.HOURS)
                .setConstraints(new Constraints.Builder()
                        .setRequiredNetworkType(NetworkType.CONNECTED)
                        .setRequiresBatteryNotLow(true)
                        .build())
                .build();

        workManager.enqueueUniquePeriodicWork(
                "sync_work",
                ExistingPeriodicWorkPolicy.REPLACE,
                workRequest);
    }
}

// Design progressive background work strategy for Android 13
private void scheduleAdaptiveBackgroundWork() {
    WorkManager workManager = WorkManager.getInstance(this);
```

```
    long lastUsedTimestamp = getLastUsedTimestamp();
    long currentTime = System.currentTimeMillis();
    long daysSinceLastUse = (currentTime - lastUsedTimestamp) / (24 * 60 * 60 * 1000);

    if (daysSinceLastUse < 1) {
        // Actively active: frequent sync
        scheduleFrequentSync();
    } else if (daysSinceLastUse < 7) {
        // Moderately active: moderate sync
        scheduleModerateSync();
    } else {
        // Inactive: minimal sync
        scheduleMinimalSync();
    }
}

private void scheduleFrequentSync() {
    WorkManager workManager = WorkManager.getInstance(this);

    PeriodicWorkRequest workRequest = new PeriodicWorkRequest.Builder(
            SyncWorker.class,
            30, TimeUnit.MINUTES)
            .setConstraints(new Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.CONNECTED)
                    .build())
            .build();

    workManager.enqueueUniquePeriodicWork(
            "sync_work",
            ExistingPeriodicWorkPolicy.REPLACE,
            workRequest);
}

private void scheduleModerateSync() {
    // Similar to scheduleFrequentSync, but with longer interval, e.g., 3 hours
```

```
}

private void scheduleMinimalSync() {
    // Similar to scheduleFrequentSync, but with longer interval, e.g., 1 day
}
```

14. App Standby Buckets Restrictions

Category: Battery Optimization

Change Content: Android 13 enhances the app standby buckets mechanism, imposing stricter limits on background app activity frequency and resource usage.

Change Date: August 2022

Reference Link: https://developer.android.com/topic/performance/power/power-details

Code Examples:

```
// Check app standby bucket status
private void checkAppStandbyBucket() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.P) {
        UsageStatsManager usageStatsManager =
                (UsageStatsManager) getSystemService(Context.USAGE_STATS_SERVICE);

        if (usageStatsManager != null) {
            int bucket = usageStatsManager.getAppStandbyBucket();
            String bucketName = getBucketName(bucket);

            Log.d("Standby", "Current app standby bucket: " + bucketName);

            // Android 13: Implement more granular adaptive strategy based on bucket status
            switch (bucket) {
                case UsageStatsManager.STANDBY_BUCKET_ACTIVE:
                    applyActiveBucketStrategy();
                    break;
```

```
            case UsageStatsManager.STANDBY_BUCKET_WORKING_SET:
                applyWorkingSetBucketStrategy();
                break;
            case UsageStatsManager.STANDBY_BUCKET_FREQUENT:
                applyFrequentBucketStrategy();
                break;
            case UsageStatsManager.STANDBY_BUCKET_RARE:
                applyRareBucketStrategy();
                break;
            case UsageStatsManager.STANDBY_BUCKET_RESTRICTED:
                applyRestrictedBucketStrategy();
                break;
            default:
                // Android 10: Simple handling for unknown buckets
                if (bucket <= UsageStatsManager.STANDBY_BUCKET_ACTIVE) {
                    scheduleFrequentTasks();
                } else {
                    scheduleInfrequentTasks();
                }
                break;
        }
    }
  }
}

// Get bucket name from bucket ID
private String getBucketName(int bucket) {
  switch (bucket) {
      case UsageStatsManager.STANDBY_BUCKET_ACTIVE:
          return "Active";
      case UsageStatsManager.STANDBY_BUCKET_WORKING_SET:
          return "Working Set";
      case UsageStatsManager.STANDBY_BUCKET_FREQUENT:
          return "Frequent";
      case UsageStatsManager.STANDBY_BUCKET_RARE:
          return "Rare";
```

```
        case UsageStatsManager.STANDBY_BUCKET_RESTRICTED:
            return "Restricted";
        default:
            return "Unknown (" + bucket + ")";
    }
}

// Android 13: Active bucket strategy
private void applyActiveBucketStrategy() {
    preloadContent();
    WorkManager workManager = WorkManager.getInstance(this);

    PeriodicWorkRequest syncRequest = new PeriodicWorkRequest.Builder(
            SyncWorker.class,
            15, TimeUnit.MINUTES)
            .setConstraints(new Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.CONNECTED)
                    .build())
            .build();

    workManager.enqueueUniquePeriodicWork(
            "active_sync",
            ExistingPeriodicWorkPolicy.REPLACE,
            syncRequest);

    enableFrequentNotifications();
}

// Android 13: Working set bucket strategy
private void applyWorkingSetBucketStrategy() {
    WorkManager workManager = WorkManager.getInstance(this);

    PeriodicWorkRequest syncRequest = new PeriodicWorkRequest.Builder(
            SyncWorker.class,
            1, TimeUnit.HOURS)
            .setConstraints(new Constraints.Builder()
```

```
                .setRequiredNetworkType(NetworkType.CONNECTED)
                .build())
        .build();

    workManager.enqueueUniquePeriodicWork(
        "working_set_sync",
        ExistingPeriodicWorkPolicy.REPLACE,
        syncRequest);

    loadContentOnDemand();
    enableModerateNotifications();
}

// Android 13: Frequent bucket strategy
private void applyFrequentBucketStrategy() {
    WorkManager workManager = WorkManager.getInstance(this);

    PeriodicWorkRequest syncRequest = new PeriodicWorkRequest.Builder(
        SyncWorker.class,
        6, TimeUnit.HOURS)
        .setConstraints(new Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresBatteryNotLow(true)
            .build())
        .build();

    workManager.enqueueUniquePeriodicWork(
        "frequent_sync",
        ExistingPeriodicWorkPolicy.REPLACE,
        syncRequest);

    reduceCacheSize();
    enableMinimalNotifications();
}

// Android 13: Rare bucket strategy
```

```
private void applyRareBucketStrategy() {
    WorkManager workManager = WorkManager.getInstance(this);

    PeriodicWorkRequest syncRequest = new PeriodicWorkRequest.Builder(
            SyncWorker.class,
            24, TimeUnit.HOURS)
            .setConstraints(new Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.UNMETERED)
                    .setRequiresBatteryNotLow(true)
                    .setRequiresDeviceIdle(true)
                    .build())
            .build();

    workManager.enqueueUniquePeriodicWork(
            "rare_sync",
            ExistingPeriodicWorkPolicy.REPLACE,
            syncRequest);

    clearCache();
    disableNonEssentialNotifications();
}

// Android 13: Restricted bucket strategy
private void applyRestrictedBucketStrategy() {
    WorkManager.getInstance(this).cancelAllWork();
    clearAllNonEssentialResources();
    disableAllNotifications();
    prepareForHibernation();
}

// Android 10: Schedule frequent tasks
private void scheduleFrequentTasks() {
    AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
    // Implement frequent task scheduling
}
```

```
// Android 10: Schedule infrequent tasks
private void scheduleInfrequentTasks() {
    AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
    // Implement infrequent task scheduling
}
```

15. Camera and Microphone Indicators

Category: Privacy and Security

Change Content: Android 13 displays status bar indicators when camera or microphone is in use. Apps need to ensure proper resource release to avoid persistent indicators.

Change Date: August 2022

Reference Link: https://developer.android.com/training/camera-deprecated

Code Examples:

```
public class CameraActivity extends AppCompatActivity {
    // Android 10: Camera API
    private Camera camera;
    private CameraPreview cameraPreview;

    // Android 13: CameraX API
    private ProcessCameraProvider cameraProvider;
    private Preview preview;
    private ImageCapture imageCapture;
    private Camera cameraX;
    private PreviewView viewFinder;
    private ExecutorService cameraExecutor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_camera);
```

```
    // Android 13: Initialize viewFinder for CameraX
    viewFinder = findViewById(R.id.viewFinder);

    // Request camera permission
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
            != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.CAMERA},
                REQUEST_CAMERA_PERMISSION);
    } else {
        // Android 13: Start CameraX
        startCameraX();

        // Android 10: Get camera instance
        camera = getCameraInstance();
        if (camera != null) {
            cameraPreview = new CameraPreview(this, camera);
            FrameLayout preview = findViewById(R.id.camera_preview);
            preview.addView(cameraPreview);
        }
    }

    // Android 13: Set up capture button
    findViewById(R.id.camera_capture_button).setOnClickListener(v -> {
        takePhoto();
    });

    // Android 13: Initialize camera executor
    cameraExecutor = Executors.newSingleThreadExecutor();
}

// Android 10: Get camera instance
private Camera getCameraInstance() {
    Camera c = null;
    try {
```

```
        c = Camera.open();
    } catch (Exception e) {
        Log.e("Camera", "Failed to get camera", e);
    }
    return c;
}

// Android 13: Start CameraX
private void startCameraX() {
    ListenableFuture<ProcessCameraProvider> cameraProviderFuture =
            ProcessCameraProvider.getInstance(this);

    cameraProviderFuture.addListener(() -> {
        try {
            cameraProvider = cameraProviderFuture.get();
            preview = new Preview.Builder().build();
            imageCapture = new ImageCapture.Builder()
                    .setCaptureMode(ImageCapture.CAPTURE_MODE_MINIMIZE_LATENCY)
                    .build();

            CameraSelector cameraSelector = CameraSelector.DEFAULT_BACK_CAMERA;
            cameraProvider.unbindAll();
            cameraX = cameraProvider.bindToLifecycle(
                    this, cameraSelector, preview, imageCapture);

            preview.setSurfaceProvider(viewFinder.getSurfaceProvider());

        } catch (ExecutionException | InterruptedException e) {
            Log.e("Camera", "Failed to bind use cases", e);
        }
    }, ContextCompat.getMainExecutor(this));
}

// Android 13: Take photo using CameraX
private void takePhoto() {
    if (imageCapture == null) {
```

```
        return;
    }

    File photoFile = new File(getOutputDirectory(),
            new SimpleDateFormat("yyyy-MM-dd-HH-mm-ss-SSS", Locale.getDefault())
                    .format(System.currentTimeMillis()) + ".jpg");

    ImageCapture.OutputFileOptions outputOptions =
            new ImageCapture.OutputFileOptions.Builder(photoFile).build();

    imageCapture.takePicture(
            outputOptions,
            ContextCompat.getMainExecutor(this),
            new ImageCapture.OnImageSavedCallback() {
                @Override
                public void onImageSaved(ImageCapture.OutputFileResults outputFileResults) {
                    String msg = "Photo saved successfully: " + photoFile.getAbsolutePath();
                    Toast.makeText(CameraActivity.this, msg, Toast.LENGTH_SHORT).show();
                    Log.d("Camera", msg);
                }

                @Override
                public void onError(ImageCaptureException exception) {
                    Log.e("Camera", "Failed to save photo", exception);
                }
            });
}

// Android 13: Get output directory for photos
private File getOutputDirectory() {
    File mediaDir = getExternalMediaDirs()[0];
    File appDir = new File(mediaDir, getResources().getString(R.string.app_name));
    if (!appDir.exists() && !appDir.mkdirs()) {
        appDir = getFilesDir();
    }
    return appDir;
```

```
}

@Override
protected void onPause() {
    super.onPause();
    // Android 10: Release camera resources
    releaseCamera();

    // Android 13: Unbind all use cases
    if (cameraProvider != null) {
        cameraProvider.unbindAll();
    }
}

@Override
protected void onResume() {
    super.onResume();
    // Android 13: Restart CameraX
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
            == PackageManager.PERMISSION_GRANTED) {
        startCameraX();
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // Android 13: Shutdown camera executor
    cameraExecutor.shutdown();

    // Android 13: Ensure camera resources are released
    if (cameraProvider != null) {
        cameraProvider.unbindAll();
    }
}
```

```
    // Android 10: Release camera resources
    private void releaseCamera() {
        if (camera != null) {
            camera.release();
            camera = null;
        }
    }
}
```

16. Background Sensor Access Restrictions

Category: Privacy and Security

Change Content: Android 13 restricts apps' ability to access sensors in the background, requiring specific permissions and following usage rules.

Change Date: August 2022

Reference Link: https://developer.android.com/guide/topics/sensors/sensors_overview

Code Examples:

```
public class SensorActivity extends AppCompatActivity implements SensorEventListener {
    private SensorManager sensorManager;
    private Sensor accelerometer;
    private boolean isRegistered = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sensor);

        // Get sensor manager
        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

        // Get accelerometer sensor
```

```
    accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

    // Register sensor listener
    if (accelerometer != null) {
        sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
        isRegistered = true;
        Log.d("Sensor", "Sensor registered");
    } else {
        Log.e("Sensor", "This device has no accelerometer");
        Toast.makeText(this, "This device doesn't support accelerometer", Toast.LENGTH_SHORT).show();
    }
}

@Override
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];

        Log.d("Sensor", "Acceleration: x=" + x + ", y=" + y + ", z=" + z);

        runOnUiThread(() -> {
            processAccelerometerData(x, y, z);
        });
    }
}

private void processAccelerometerData(float x, float y, float z) {
    TextView tvX = findViewById(R.id.tv_x_value);
    TextView tvY = findViewById(R.id.tv_y_value);
    TextView tvZ = findViewById(R.id.tv_z_value);

    tvX.setText(String.format("X: %.2f", x));
    tvY.setText(String.format("Y: %.2f", y));
    tvZ.setText(String.format("Z: %.2f", z));
```

```
    double magnitude = Math.sqrt(x*x + y*y + z*z);
    if (magnitude > 15) {
        Toast.makeText(this, "Intense motion detected!", Toast.LENGTH_SHORT).show();
    }
}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    Log.d("Sensor", "Sensor accuracy changed: " + accuracy);
}

@Override
protected void onPause() {
    super.onPause();
    // Unregister sensor listener when app goes to background
    if (isRegistered) {
        sensorManager.unregisterListener(this);
        isRegistered = false;
        Log.d("Sensor", "Sensor unregistered in onPause");
    }
}

@Override
protected void onResume() {
    super.onResume();
    // Re-register sensor listener when app comes to foreground
    if (!isRegistered && sensorManager != null && accelerometer != null) {
        // Check permissions
        if ((Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU &&
                ContextCompat.checkSelfPermission(this, Manifest.permission.BODY_SENSORS)
                        == PackageManager.PERMISSION_GRANTED) ||
            Build.VERSION.SDK_INT < Build.VERSION_CODES.TIRAMISU) {

            sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
            isRegistered = true;
```

```
            Log.d("Sensor", "Sensor re-registered in onResume");
        }
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // Ensure sensor listener is unregistered when Activity is destroyed
    if (isRegistered) {
        sensorManager.unregisterListener(this);
        isRegistered = false;
        Log.d("Sensor", "Sensor unregistered in onDestroy");
    }
}

// Start foreground service for background sensor access
public void startSensorForegroundService() {
    Intent serviceIntent = new Intent(this, SensorForegroundService.class);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        startForegroundService(serviceIntent);
    } else {
        startService(serviceIntent);
    }
}

// Foreground service example
public static class SensorForegroundService extends Service implements SensorEventListener {
    private SensorManager sensorManager;
    private Sensor accelerometer;
    private static final int NOTIFICATION_ID = 1001;
    private static final String CHANNEL_ID = "sensor_channel";

    @Override
    public void onCreate() {
```

```
    super.onCreate();

    createNotificationChannel();
    Notification notification = createNotification();
    startForeground(NOTIFICATION_ID, notification);

    sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

    if (checkSensorPermissions()) {
        sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
        Log.d("SensorService", "Sensor registered in foreground service");
    } else {
        Log.e("SensorService", "Insufficient permissions to use sensors");
        stopSelf();
    }
}

private boolean checkSensorPermissions() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        return ContextCompat.checkSelfPermission(this, Manifest.permission.BODY_SENSORS)
                == PackageManager.PERMISSION_GRANTED;
    } else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
        return ContextCompat.checkSelfPermission(this, Manifest.permission.ACTIVITY_RECOGNITION)
                == PackageManager.PERMISSION_GRANTED;
    }
    return true;
}

private void createNotificationChannel() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationChannel channel = new NotificationChannel(
                CHANNEL_ID,
                "Sensor Service",
                NotificationManager.IMPORTANCE_LOW);
        channel.setDescription("Notification for monitoring sensor data in background");
```

```
        NotificationManager notificationManager = getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
}

private Notification createNotification() {
    Intent notificationIntent = new Intent(this, SensorActivity.class);
    PendingIntent pendingIntent = PendingIntent.getActivity(
            this, 0, notificationIntent, PendingIntent.FLAG_IMMUTABLE);

    return new NotificationCompat.Builder(this, CHANNEL_ID)
            .setContentTitle("Sensor Monitoring")
            .setContentText("Monitoring sensor data in background")
            .setSmallIcon(R.drawable.ic_sensor)
            .setContentIntent(pendingIntent)
            .build();
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    return START_STICKY;
}

@Override
public IBinder onBind(Intent intent) {
    return null;
}

@Override
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];
```

```
        Log.d("SensorService", "Acceleration: x=" + x + ", y=" + y + ", z=" + z);
        processBackgroundSensorData(x, y, z);
    }
}

private void processBackgroundSensorData(float x, float y, float z) {
    double magnitude = Math.sqrt(x*x + y*y + z*z);
    if (magnitude < 2) {
        sendFallDetectionNotification();
    }
}

private void sendFallDetectionNotification() {
    String channelId = "fall_detection_channel";
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationChannel channel = new NotificationChannel(
                channelId,
                "Fall Detection",
                NotificationManager.IMPORTANCE_HIGH);
        NotificationManager notificationManager = getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }

    Intent intent = new Intent(this, SensorActivity.class);
    PendingIntent pendingIntent = PendingIntent.getActivity(
            this, 0, intent, PendingIntent.FLAG_IMMUTABLE);

    Notification notification = new NotificationCompat.Builder(this, channelId)
            .setContentTitle("Possible Fall Detected")
            .setContentText("We detected a possible fall, tap for details")
            .setSmallIcon(R.drawable.ic_fall_detection)
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setContentIntent(pendingIntent)
            .setAutoCancel(true)
            .build();
```

```
        NotificationManager notificationManager =
                (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(NOTIFICATION_ID, notification);
    }
  }
}
```

17. Near Field Communication Restrictions

Category: Connectivity

Change Content: Android 13 imposes stricter permission controls on NFC functionality, requiring apps to properly declare permissions and handle permission requests.

Change Date: August 2022

Reference Link: https://developer.android.com/guide/topics/connectivity/nfc/advanced-nfc

Code Examples:

```
public class NfcActivity extends AppCompatActivity {
  private NfcAdapter nfcAdapter;
  private PendingIntent pendingIntent;
  private static final int REQUEST_NFC_PERMISSION = 100;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_nfc);

    // Android 13: Check NFC permission
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU &&
        ContextCompat.checkSelfPermission(this, Manifest.permission.NFC)
        != PackageManager.PERMISSION_GRANTED) {
      ActivityCompat.requestPermissions(this,
          new String[]{Manifest.permission.NFC},
```

```
                REQUEST_NFC_PERMISSION);
        return;
    }

    initializeNfc();
}

private void initializeNfc() {
    // Get NFC adapter
    nfcAdapter = NfcAdapter.getDefaultAdapter(this);
    if (nfcAdapter == null) {
        Toast.makeText(this, "This device doesn't support NFC", Toast.LENGTH_SHORT).show();
        finish();
        return;
    }

    // Check if NFC is enabled
    if (!nfcAdapter.isEnabled()) {
        // Android 13: Use AlertDialog for NFC settings prompt
        new AlertDialog.Builder(this)
                .setTitle("NFC is Disabled")
                .setMessage("This feature requires NFC, please enable it in settings")
                .setPositiveButton("Go to Settings", (dialog, which) -> {
                    startActivity(new Intent(Settings.ACTION_NFC_SETTINGS));
                })
                .setNegativeButton("Cancel", (dialog, which) -> finish())
                .show();
        return;
    }

    // Create PendingIntent
    pendingIntent = PendingIntent.getActivity(this, 0,
            new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP),
            Build.VERSION.SDK_INT >= Build.VERSION_CODES.M ? PendingIntent.FLAG_IMMUTABLE :
0);
```

```
    // Handle Intent
    handleIntent(getIntent());
}

@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == REQUEST_NFC_PERMISSION) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // Permission granted, initialize NFC
            initializeNfc();
        } else {
            // Permission denied
            Toast.makeText(this, "NFC permission is needed for this feature", Toast.LENGTH_SHORT).show();
            finish();
        }
    }
}

@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    handleIntent(intent);
}

private void handleIntent(Intent intent) {
    // Handle NFC tag
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction()) ||
            NfcAdapter.ACTION_TECH_DISCOVERED.equals(intent.getAction()) ||
            NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {

        Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
        if (tag != null) {
            showNfcTagInfo(tag);
```

```
            Ndef ndef = Ndef.get(tag);
            if (ndef != null) {
                readNfcTag(ndef);
            } else {
                Toast.makeText(this, "Unsupported NFC tag format", Toast.LENGTH_SHORT).show();
            }
        }
    }
}

private void showNfcTagInfo(Tag tag) {
    StringBuilder info = new StringBuilder();
    byte[] tagId = tag.getId();
    info.append("Tag ID: ").append(bytesToHex(tagId)).append("\n");
    info.append("Supported technologies: \n");
    for (String tech : tag.getTechList()) {
        info.append("- ").append(tech.substring(tech.lastIndexOf(".") + 1)).append("\n");
    }

    TextView tagInfoView = findViewById(R.id.text_tag_info);
    tagInfoView.setText(info.toString());
}

private String bytesToHex(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (byte b : bytes) {
        sb.append(String.format("%02X ", b));
    }
    return sb.toString();
}

private void readNfcTag(Ndef ndef) {
    try {
        ndef.connect();
        NdefMessage ndefMessage = ndef.getNdefMessage();
        if (ndefMessage != null) {
```

```
            NdefRecord[] records = ndefMessage.getRecords();
            StringBuilder content = new StringBuilder();

            for (NdefRecord record : records) {
                if (record.getTnf() == NdefRecord.TNF_WELL_KNOWN &&
                        Arrays.equals(record.getType(), NdefRecord.RTD_TEXT)) {
                    // Process text record
                    byte[] payload = record.getPayload();
                    String textEncoding = ((payload[0] & 0x80) == 0) ? "UTF-8" : "UTF-16";
                    int languageCodeLength = payload[0] & 0x3F;
                    String text = new String(payload, languageCodeLength + 1,
                            payload.length - languageCodeLength - 1, textEncoding);
                    content.append("Text: ").append(text).append("\n");
                } else if (record.getTnf() == NdefRecord.TNF_WELL_KNOWN &&
                        Arrays.equals(record.getType(), NdefRecord.RTD_URI)) {
                    // Process URI record
                    byte[] payload = record.getPayload();
                    int prefixCode = payload[0] & 0xFF;
                    String prefix = NFC_URI_PREFIX[prefixCode];
                    String uri = prefix + new String(payload, 1, payload.length - 1, "UTF-8");
                    content.append("URI: ").append(uri).append("\n");
                }
            }

            displayNfcContent(content.toString());
        } else {
            displayNfcContent("NFC tag has no NDEF message");
        }
        ndef.close();
    } catch (Exception e) {
        Log.e("NFC", "Failed to read NFC tag", e);
        displayNfcContent("Read failed: " + e.getMessage());
    }
}

// URI prefix list
```

```
    private static final String[] NFC_URI_PREFIX = {
            "", "http://www.", "https://www.", "http://", "https://", "tel:", "mailto:",
            "ftp://anonymous:anonymous@", "ftp://ftp.", "ftps://", "sftp://", "smb://",
            "nfs://", "ftp://", "dav://", "news:", "telnet://", "imap:", "rtsp://", "urn:",
            "pop:", "sip:", "sips:", "tftp:", "btspp://", "btl2cap://", "btgoep://", "tcpobex://",
            "irdaobex://", "file://", "urn:epc:id:", "urn:epc:tag:", "urn:epc:pat:", "urn:epc:raw:",
            "urn:epc:", "urn:nfc:"
    };

    private void displayNfcContent(String content) {
        TextView textView = findViewById(R.id.text_nfc_content);
        textView.setText(content);
    }

    @Override
    protected void onResume() {
        super.onResume();
        // Enable foreground dispatch
        if (nfcAdapter != null) {
            nfcAdapter.enableForegroundDispatch(this, pendingIntent, null, null);
        }
    }

    @Override
    protected void onPause() {
        super.onPause();
        // Disable foreground dispatch
        if (nfcAdapter != null) {
            nfcAdapter.disableForegroundDispatch(this);
        }
    }
}
```

18. Audio Focus Management Changes

Category: Media

Change Content: Android 13 improves audio focus management, requiring apps to adapt to new focus rules to avoid abnormal audio playback behavior.

Change Date: August 2022

Reference Link: https://developer.android.com/guide/topics/media-apps/audio-focus

Code Examples:

```
public class AudioPlayerActivity extends AppCompatActivity {
    private MediaPlayer mediaPlayer;
    private AudioManager audioManager;
    private AudioAttributes audioAttributes;
    private AudioFocusRequest audioFocusRequest;
    private boolean playbackDelayed = false;
    private boolean resumeOnFocusGain = false;
    private boolean playbackNowAuthorized = false;

    // Audio focus change listener
    private final AudioManager.OnAudioFocusChangeListener focusChangeListener =
            new AudioManager.OnAudioFocusChangeListener() {
        @Override
        public void onAudioFocusChange(int focusChange) {
            switch (focusChange) {
                case AudioManager.AUDIOFOCUS_GAIN:
                    // Gained audio focus
                    Log.d("AudioFocus", "Gained audio focus");
                    if (playbackDelayed || resumeOnFocusGain) {
                        playbackDelayed = false;
                        resumeOnFocusGain = false;
                        playbackNowAuthorized = true;
                        if (mediaPlayer != null) {
```

```
                mediaPlayer.start();
            }
        }
        break;
    case AudioManager.AUDIOFOCUS_LOSS:
        // Permanent loss of audio focus
        Log.d("AudioFocus", "Permanently lost audio focus");
        resumeOnFocusGain = false;
        playbackDelayed = false;
        playbackNowAuthorized = false;
        if (mediaPlayer != null && mediaPlayer.isPlaying()) {
            mediaPlayer.pause();
            // On Android 13, might need to completely stop playback
            mediaPlayer.stop();
            try {
                mediaPlayer.prepare();
            } catch (IOException e) {
                Log.e("AudioPlayer", "MediaPlayer preparation failed", e);
            }
        }
        break;
    case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT:
        // Temporary loss of audio focus
        Log.d("AudioFocus", "Temporarily lost audio focus");
        resumeOnFocusGain = mediaPlayer != null && mediaPlayer.isPlaying();
        if (mediaPlayer != null && mediaPlayer.isPlaying()) {
            mediaPlayer.pause();
        }
        break;
    case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK:
        // Temporary loss of audio focus, but can continue playing at lower volume
        Log.d("AudioFocus", "Temporarily lost audio focus, can duck");
        if (mediaPlayer != null && mediaPlayer.isPlaying()) {
            // On Android 13, recommended to pause when ducked instead of lowering volume
            mediaPlayer.pause();
            resumeOnFocusGain = true;
```

```
            }
            break;
        }
    }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_audio_player);

    audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

    // Create audio attributes
    audioAttributes = new AudioAttributes.Builder()
            .setUsage(AudioAttributes.USAGE_MEDIA)
            .setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
            .build();

    // Create audio focus request
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        audioFocusRequest = new AudioFocusRequest.Builder(AudioManager.AUDIOFOCUS_GAIN)
                .setAudioAttributes(audioAttributes)
                .setAcceptsDelayedFocusGain(true)
                .setWillPauseWhenDucked(true) // On Android 13, recommended to pause when ducked
                .setOnAudioFocusChangeListener(focusChangeListener)
                .build();
    }

    // Initialize MediaPlayer
    mediaPlayer = new MediaPlayer();
    mediaPlayer.setAudioAttributes(audioAttributes);

    try {
        Uri uri = Uri.parse("https://example.com/audio.mp3");
        mediaPlayer.setDataSource(this, uri);
```

```
            mediaPlayer.prepare();

            // Set completion listener
            mediaPlayer.setOnCompletionListener(mp -> {
                // Abandon audio focus
                abandonAudioFocus();
                playbackNowAuthorized = false;

                // Update UI
                updatePlaybackUI(false);
            });
        } catch (IOException e) {
            Log.e("AudioPlayer", "MediaPlayer preparation failed", e);
        }

        findViewById(R.id.btn_play).setOnClickListener(v -> {
            playAudio();
        });

        findViewById(R.id.btn_pause).setOnClickListener(v -> {
            pauseAudio();
        });

        findViewById(R.id.btn_stop).setOnClickListener(v -> {
            stopAudio();
        });
    }

    private void playAudio() {
        if (!playbackNowAuthorized) {
            // Request audio focus
            int result;
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                result = audioManager.requestAudioFocus(audioFocusRequest);
            } else {
                result = audioManager.requestAudioFocus(focusChangeListener,
```

```
                AudioManager.STREAM_MUSIC,
                AudioManager.AUDIOFOCUS_GAIN);
        }

        switch (result) {
            case AudioManager.AUDIOFOCUS_REQUEST_GRANTED:
                // Focus granted, start playback
                Log.d("AudioFocus", "Audio focus granted, starting playback");
                playbackNowAuthorized = true;
                mediaPlayer.start();
                updatePlaybackUI(true);
                break;
            case AudioManager.AUDIOFOCUS_REQUEST_DELAYED:
                // Delayed focus gain
                Log.d("AudioFocus", "Audio focus request delayed");
                playbackDelayed = true;
                Toast.makeText(this, "Playback will start shortly", Toast.LENGTH_SHORT).show();
                break;
            case AudioManager.AUDIOFOCUS_REQUEST_FAILED:
                // Focus not granted
                Log.d("AudioFocus", "Failed to get audio focus");
                playbackNowAuthorized = false;
                Toast.makeText(this, "Cannot get audio focus", Toast.LENGTH_SHORT).show();
                break;
        }
    } else if (!mediaPlayer.isPlaying()) {
        // Have focus but paused, resume playback
        mediaPlayer.start();
        updatePlaybackUI(true);
    }
}

private void pauseAudio() {
    if (mediaPlayer != null && mediaPlayer.isPlaying()) {
        mediaPlayer.pause();
        updatePlaybackUI(false);
```

```
    }
}

private void stopAudio() {
    if (mediaPlayer != null) {
        if (mediaPlayer.isPlaying()) {
            mediaPlayer.stop();
            try {
                mediaPlayer.prepare();
            } catch (IOException e) {
                Log.e("AudioPlayer", "MediaPlayer preparation failed", e);
            }
            updatePlaybackUI(false);
        }

        // Abandon audio focus
        abandonAudioFocus();
        playbackNowAuthorized = false;
    }
}

private void abandonAudioFocus() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        if (audioFocusRequest != null) {
            audioManager.abandonAudioFocusRequest(audioFocusRequest);
        }
    } else {
        audioManager.abandonAudioFocus(focusChangeListener);
    }
}

private void updatePlaybackUI(boolean isPlaying) {
    findViewById(R.id.btn_play).setEnabled(!isPlaying);
    findViewById(R.id.btn_pause).setEnabled(isPlaying);
    findViewById(R.id.btn_stop).setEnabled(isPlaying);
}
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mediaPlayer != null) {
            if (mediaPlayer.isPlaying()) {
                mediaPlayer.stop();
            }
            mediaPlayer.release();
            mediaPlayer = null;
        }

        abandonAudioFocus();
    }
}
```

19. JobScheduler Restrictions

Category: Background Processing

Change Content: Android 13 imposes stricter restrictions on JobScheduler, including stricter execution windows and lower running frequency.

Change Date: August 2022

Reference Link: https://developer.android.com/reference/android/app/job/JobScheduler

Code Examples:

```
// Schedule frequent background tasks
private void scheduleJob() {
    JobScheduler jobScheduler = (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);

    ComponentName serviceName = new ComponentName(this, MyJobService.class);
    JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName)
            .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
```

```
            .setPeriodic(15 * 60 * 1000) // Execute every 15 minutes
            .setPersisted(true);

    int resultCode = jobScheduler.schedule(builder.build());
    if (resultCode == JobScheduler.RESULT_SUCCESS) {
        Log.d("JobScheduler", "Job scheduled successfully");
    } else {
        Log.d("JobScheduler", "Job scheduling failed");
    }
}
```

20. Non-SDK Interface Restrictions

Category: API Compatibility

Change Content: Android 13 further restricts access to non-SDK interfaces, with more previously available non-public APIs being added to the greylist or blacklist.

Change Date: August 2022

Reference Link: https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces

Code Examples:

```
// Attempt to use reflection to access non-SDK interfaces (Android 10)
private void useHiddenApis() {
    try {
        // Reflect to access hidden API
        Class<?> activityManagerClass = Class.forName("android.app.ActivityManager");
        Method getServiceMethod = activityManagerClass.getDeclaredMethod("getService");
        getServiceMethod.setAccessible(true);
        Object activityManagerService = getServiceMethod.invoke(null);

        // Further use hidden service interface
        Class<?> iActivityManagerClass = Class.forName("android.app.IActivityManager");
```

```
        Method getProcessPssMethod = iActivityManagerClass.getDeclaredMethod("getProcessPss", int[].class);
        getProcessPssMethod.setAccessible(true);
        long[] pss = (long[]) getProcessPssMethod.invoke(activityManagerService, new int[]{Process.myPid()});

        Log.d("HiddenAPI", "Get PSS using hidden API: " + Arrays.toString(pss));
    } catch (Exception e) {
        Log.e("HiddenAPI", "Failed to access hidden API", e);
    }
}

// Check non-SDK interface access restrictions and use public APIs instead (Android 13)
private void usePublicApis() {
    // Use official public APIs instead of hidden APIs
    ActivityManager activityManager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);

    // Get memory information
    ActivityManager.MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();
    activityManager.getMemoryInfo(memoryInfo);

    Log.d("PublicAPI", "Available memory: " + memoryInfo.availMem / (1024 * 1024) + " MB");
    Log.d("PublicAPI", "Total memory: " + memoryInfo.totalMem / (1024 * 1024) + " MB");

    // Get statistics for running processes
    List<ActivityManager.RunningAppProcessInfo> runningAppProcesses =
            activityManager.getRunningAppProcesses();
    if (runningAppProcesses != null) {
        for (ActivityManager.RunningAppProcessInfo processInfo : runningAppProcesses) {
            if (processInfo.pid == Process.myPid()) {
                // Get memory usage for current process
                int[] pids = {processInfo.pid};
                Debug.MemoryInfo[] memoryInfoArray = activityManager.getProcessMemoryInfo(pids);
                if (memoryInfoArray.length > 0) {
                    Debug.MemoryInfo processMemoryInfo = memoryInfoArray[0];
                    Log.d("PublicAPI", "Process PSS: " + processMemoryInfo.getTotalPss() + " KB");
                }
                break;
```

```
        }
      }
    }

    // Detect restricted API usage
    detectRestrictedApiUsage();
}

// Detect usage of restricted APIs
private void detectRestrictedApiUsage() {
    StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
            .detectNonSdkApiUsage()
            .penaltyLog()
            .build());
}
```