DAYANANDA SAGAR COLLEGE OF ENGG.

(An Autonomous Institute Affiliated to Visvesvaraya Technological University, Belagavi & Approved by AICTE, New Delhi.)

Department of Computer Science And Engg.

Subject: Mobile Application Development(MAD)

By

Dr. Rohini T V,

Associate Professor

V: A & B Section

Department of Computer Science & Engineering

Aca. Year ODD SEM /2022-23

Module-3

In this topic students will be learning :

- Working in the Background: Background Tasks, Async Task and Async Task Loader, Connect to the Internet, Broadcast Receivers, Services
- Triggering: Scheduling
- optimizing background tasks: Notifications, Scheduling Alarms

Topic Name

- 3.1 Working in the Background: Background Tasks, Async Task and Async Task Loader, Connect to the Internet, Broadcast Receivers, Services
- 3.2 Triggering: Scheduling
- 3.3 optimizing background tasks: Notifications, Scheduling Alarms

Working in the Background

There are two ways to do background processing in Android:

- Using the AsyncTask class
- Using the Loader framework

In most situations you'll choose the Loader framework, but it's important to know how AsyncTask works so you can make a good choice.

The UI thread

- When an Android app starts, it creates the main thread, which is often called the UI thread.
- The UI thread dispatches events to the appropriate user interface (UI) widgets, and it's where your app interacts with components from the Android UI toolkit (components from the android.widget and android.view packages).

Android's thread model has two rules:

1. Do not block the UI thread.

The UI thread needs to give its attention to drawing the UI and keeping the app responsive to user input. If everything happened on the UI thread, long operations such as network access or database queries could block the whole UI.

To make sure your app doesn't block the UI thread:

Complete all work in less than 16 ms for each UI screen.Don't run asynchronous tasks and other long-running tasks on the UI thread. Instead, implement tasks on a background thread using AsyncTask (for short or interruptible tasks) or AsyncTaskLoader (for tasks that are high priority, or tasks that need to report back to the user or UI).

2. Do UI work only on the UI thread.

Don't use a background thread to manipulate your UI, because the Android UI toolkit is not thread-safe.

AsyncTask

Use the AsyncTask class to implement an asynchronous, long-running task on a worker thread. (A worker thread is any thread which is not the main or UI thread.) AsyncTask allows you to perform background operations and publish results on the UI thread without manipulating threads or handlers.

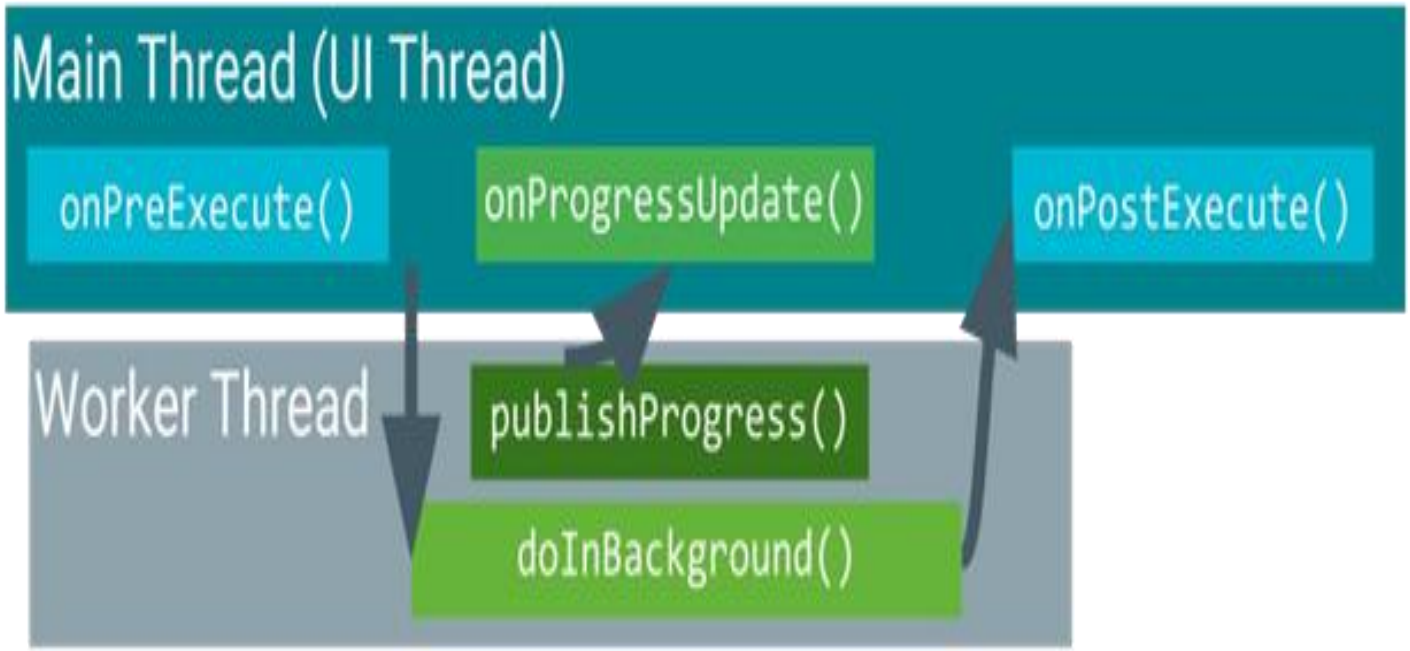
When AsyncTask is executed, it goes through four steps:

1. onPreExecute() is invoked on the UI thread before the task is executed. This step is normally used to set up the task, for instance by showing a progress bar in the UI.

2. doInBackground(Params...) is invoked on the background thread immediately after onPreExecute() finishes. This step performs a background computation, returns a result, and passes the result to onPostExecute() . The doInBackground() method can also call publishProgress(Progress...) to publish one or more units of progress.

3. onProgressUpdate(Progress...) runs on the UI thread after publishProgress(Progress...) is invoked. Use onProgressUpdate() to report any form of progress to the UI thread while the background computation is executing.For instance, you can use it to pass the data to animate a progress bar or show logs in a text field.

4. onPostExecute(Result) runs on the UI thread after the background computation has finished.


Main Thread (UI Thread)
onPreExecute()
onProgressUpdate()
onPostExecute()
Worker Thread
publishProgress()
doInBackground()

AsyncTask usage

To use the AsyncTask class, define a subclass of AsyncTask that overrides the doInBackground(Params…) method (and usually the onPostExecute(Result) method as well). This section describes the parameters and usage of AsyncTask , then shows a complete example.

AsyncTask parameters

In your subclass of AsyncTask , provide the data types for three kinds of parameters:

"**Params**" specifies the type of parameters passed to doInBackground() as an array.

"**Progress**" specifies the type of parameters passed to publishProgress() on the background thread. These parameters are then passed to the onProgressUpdate() method on the main thread.

"**Result**" specifies the type of parameter that doInBackground() returns. This parameter is automatically passed to onPostExecute() on the main thread.

Limitations of AsyncTask

AsyncTask is impractical for some use cases:

- Changes to device configuration cause problems :When device configuration changes while an AsyncTask is running, for example if the user changes the screen orientation, the activity that created the AsyncTask is destroyed and re-created. The AsyncTask is unable to access the newly created activity, and the results of the AsyncTask aren't published.

- Old AsyncTask objects stay around, and your app may run out of memory or crash : If the activity that created the AsyncTask is destroyed, the AsyncTask is not destroyed along with it. For example, if your user exits the application after the AsyncTask has started, the AsyncTask keeps using resources unless you call cancel() .

Loaders

- Background tasks are commonly used to load data such as forecast reports or movie reviews.
- Loading data can be memory intensive, and you want the data to be available even if the device configuration changes.
- For these situations, use loaders, which are a set of classes that facilitate loading data into an activity.
- Loaders use the LoaderManager class to manage one or more loaders.
- LoaderManager includes a set of callbacks for when the loader is created, when it's done loading data, and when it's reset.

Starting a loader

Use the LoaderManager class to manage one or more Loader instances within an activity or fragment. Use initLoader() to initialize a loader and make it active. Typically, you do this within the activity's onCreate() method.

For example:

```
// Prepare the loader. Either reconnect with an existing one,
// or start a new one.
getLoaderManager().initLoader(0, null, this);
```

If you're using the Support Library, make this call using getSupportLoaderManager() instead of getLoaderManager() . For example:

```
getSupportLoaderManager().initLoader(0, null, this);
```

Restarting a loader

- When initLoader() reuses an existing loader, it doesn't replace the data that the loader contains, but sometimes you want it to.
- For example, when you use a user query to perform a search and the user enters a new query, you want to reload the data using the new search term.
- In this situation, use the restartLoader() method and pass in the ID of the loader you want to restart. This forces another data load with new input data.

About the restartLoader() method:

- restartLoader() uses the same arguments as initLoader() .
- restartLoader() triggers the onCreateLoader() method, just as initLoader() does when creating a new loader.
- If a loader with the given ID exists, restartLoader() restarts the identified loader and replaces its data.
- If no loader with the given ID exists, restartLoader() starts a new loader.

AsyncTaskLoader

- AsyncTaskLoader is the loader equivalent of AsyncTask . AsyncTaskLoader provides a method, loadInBackground() , that runs on a separate thread.
- The results of loadInBackground() are automatically delivered to the UI thread, by way of the onLoadFinished() LoaderManager callback.

AsyncTaskLoader usage

To define a subclass of AsyncTaskLoader , create a class that extends AsyncTaskLoader<D> , where D is the data type of the data you are loading. For example, the following AsyncTaskLoader loads a list of strings:

public static class StringListLoader extends AsyncTaskLoader<List<String>> {}

Next, implement a constructor that matches the superclass implementation:

Your constructor takes the application context as an argument and passes it into a call to super() . If your loader needs additional information to perform the load, your constructor can take additional arguments. In the example shown below, the constructor takes a query term.

```
public StringListLoader(Context context, String queryString) {
super(context);
mQueryString = queryString;
}
```

Implementing the callbacks

Use the constructor in the onCreateLoader() LoaderManager callback, which is where the new loader is created. For

example, this onCreateLoader() callback uses the StringListLoader constructor defined above:

```
@Override
public Loader<List<String>> onCreateLoader(int id, Bundle args) {
return new StringListLoader(this, args.getString("queryString"));
}
```

The results of loadInBackground() are automatically passed into the onLoadFinished() callback, which is where you can display the results in the UI. For example:

```
public void onLoadFinished(Loader<List<String>> loader, List<String> data) {
mAdapter.setData(data);
}
```

The onLoaderReset() callback is only called when the loader is being destroyed, so you can leave onLoaderReset() blank most of the time, because you won't try to access the data after the loader is destroyed.

Connect to the Internet

- Most Android applications have some data that the user interacts with; it might be news articles, weather information, contacts, game data, user information, and more.
- Often, this data is provided over the network by a web API.

Network security

- Network transactions are inherently risky, because they involve transmitting data that could be private to the user.
- People are increasingly aware of these risks, especially when their devices perform network transactions, so it's very important that your app implement best practices for keeping user data secure at all times.

Security best practices for network operations:

- Use appropriate protocols for sensitive data. For example for secure web traffic, use the HttpsURLConnection subclass of HttpURLConnection .
- Use HTTPS instead of HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently connect on insecure networks such as public Wi-Fi hotspots. Consider using SSLSocketClass to implement authenticated, encrypted socket-level communication.
- Don't use localhost network ports to handle sensitive interprocess communication (IPC), because other applications on the device can access these local ports. Instead, use a mechanism that lets you use authentication, for example a Service .
- Don't trust data downloaded from HTTP or other insecure protocols. Validate input that's entered into a WebView and responses to intents that you issue against HTTP.

Including permissions in the manifest :

- Before your app can make network calls, you need to include a permission in your AndroidManifest.xml file.
- Add the following tag inside the <manifest> tag:

<*uses-permission android:name="android.permission.INTERNET" />*

- When using the network, it's a best practice to monitor the network state of the device so that you don't attempt to make network calls when the network is unavailable.
- To access the network state of the device, your app needs an additional permission:

<*uses-permission*
android:name="android.permission.ACCESS_NETWORK_STATE" />

Making an HTTP connection

- Most network-connected Android apps use HTTP and HTTPS to send and receive data over the network. For a refresher on HTTP, visit this Learn HTTP tutorial.
- Note: If a web server offers HTTPS, you should use it instead of HTTP for improved security.
- The HttpURLConnection Android client supports HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling.
- To use the HttpURLConnection client, build a URI (the request's destination).
- Then obtain a connection, send the request and any request headers, download and read the response and any response headers, and disconnect.

Broadcast Receivers

- Explicit intents are used to start specific, fully qualified activities, as well as to pass information between activities in your app.
- Implicit intents are used to start activities based on registered components that the system is aware of, for example general functionality.

Broadcast intents

- The intents you've seen up to now always resulted in an activity being launched, either a specific activity from your application or an activity from a different application that could fulfill the requested action.
- But sometimes an intent doesn't have a specific recipient, and sometimes you don't want an activity to be launched in response to an intent.
- For example, when your app receives a system intent indicating that the network state of a device has changed, you probably don't want to launch an activity, but you may want to disable some functionality of your app.

There are two types of broadcast intents:

1. Those delivered by the system (**system broadcast intents),**
2. Those that your app delivers (**custom broadcast intents**).

System broadcast intents

The system delivers a system broadcast intent when a system event occurs that might interest your app. For example: When the device boots, the system sends an ACTION_BOOT_COMPLETED system broadcast intent.

This intent contains the constant value "android.intent.action.BOOT_COMPLETED"

- When the device is connected to external power, the system sends ACTION_POWER_CONNECTED , which contains the constant value "android.intent.action.ACTION_POWER_CONNECTED" .
- When the device is disconnected from external power, the system sends ACTION_POWER_DISCONNECTED .
- When the device is low on memory, the system sends ACTION_DEVICE_STORAGE_LOW .
- This intent contains the constant
- value "android.intent.action.DEVICE_STORAGE_LOW"

Custom broadcast intents :

- Custom broadcast intents are broadcast intents that your application sends out.
- Use a custom broadcast intent when you want your app to take an action without launching an activity, for example when you want to let other apps know that data has been downloaded to the device and is available for them to use.
- To create a custom broadcast intent, create a custom intent action.
- To deliver a custom broadcast to other apps, pass the intent to sendBroadcast() , sendOrderedBroadcast() , or sendStickyBroadcast() .
- For example, the following method creates an intent and broadcasts it to all interested broadcast receivers:

```
public void sendBroadcastIntent() {
Intent intent = new Intent();
intent.setAction("com.example.myproject.ACTION_SHOW_TOAST");
sendBroadcast(intent);
}
```

Broadcast receivers :

- Broadcast intents aren't targeted at specific recipients. Instead, interested apps register a component to "listen" for these kind of intents.
- This listening component is called a broadcast receiver.
- Use broadcast receivers to respond to messages that are broadcast from other apps or from the system.

To create a broadcast receiver:

1. Define a subclass of the BroadcastReceiver class and implement the onReceive() method.

2. Register the broadcast receiver either dynamically in Java, or statically in your app's manifest file.

Steps for describing intents :

1. Define a subclass of BroadcastReceiver
2. Registering your broadcast receiver and setting intent filters.

Define a subclass of BroadcastReceiver :

To create a broadcast receiver, define a subclass of the BroadcastReceiver class.

This subclass is where intents are delivered (if they match the intent filters you set when you register the subclass, which happens in a later step).

Within your subclass definition:

- Implement the required onReceive() method.
- Include any other logic that your broadcast receiver needs.

Registering your broadcast receiver and setting intent filters.

There are two ways to register your broadcast receiver: statically in the manifest, or dynamically in your activity.

1. Static registration

- To register your broadcast receiver statically, add a <receiver> element to your AndroidManifest.xml file. Within the <receiver> element:
- Use the path to your BroadcastReceiver subclass as the android:name attribute.
- To prevent other applications from sending broadcasts to your receiver, set the optional android:exported attribute to false . This is an important security guideline.
- To specify the types of intents the component is listening for, use a nested <intent-filter> element.

2. Dynamic registration and unregistration :

You can also register a broadcast receiver dynamically, which ties its operation to the lifecycle of your activity. To register your receiver dynamically, call registerReceiver() and pass in the BroadcastReceiver object and an intent filter.

For example:

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(ACTION_SHOW_TOAST);
mReceiver = new AlarmReceiver();
registerReceiver(mReceiver, intentFilter);
```

Services :

What is a service?

A service is an application component that performs long-running operations, usually in the background. A service doesn't provide a user interface (UI). (An activity, on the other hand, provides a UI.)

A service can be started, bound, or both:

1. A started service is a service that an application component starts by calling startService() . Use started services for tasks that run in the background to perform long-running operations. Also use started services for tasks that perform work for remote processes.

2. **A bound service** is a service that an application component binds to itself by calling bindService() . Use bound services for tasks that another app component interacts with to perform interprocess communication (IPC). For example, a bound service might handle network transactions, perform file I/O, play music, or interact with a content provider.

Declaring services in the manifest :

As with activities and other components, you must declare all services in your application's manifest file.

To declare a service, add a <service> element as a child of the <application> element. For example:

```
<manifest ... >
...
<application ... >
<service android:name="ExampleService"
android:exported="false" />
...
</application>
</manifest>
```

Started services

How a service starts:

1. An application component such as an activity calls startService() and passes in an Intent . The Intent specifies the service and includes any data for the service to use.

2. The system calls the service's onCreate() method and any other appropriate callbacks on the main thread. It's up to the service to implement these callbacks with the appropriate behavior, such as creating a secondary thread in which to work.

3. The system calls the service's onStartCommand() method, passing in the Intent supplied by the client in step 1. (The client in this context is the application component that calls the service.)

Bound services

A service is "bound" when an application component binds to it by calling bindService() .

A bound service offers a client- server interface that allows components to interact with the service, send requests, and get results, sometimes using interprocess communication (IPC) to send and receive information across processes.

A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.A bound service generally does not allow components to start it by calling startService() .

Implementing a bound service

To implement a bound service, define the interface that specifies how a client can communicate with the service.

Thisinterface, which your service returns from the onBind() callback method, must be an implementation of IBinder .

To retrieve the IBinder interface, a client application component calls bindService() .

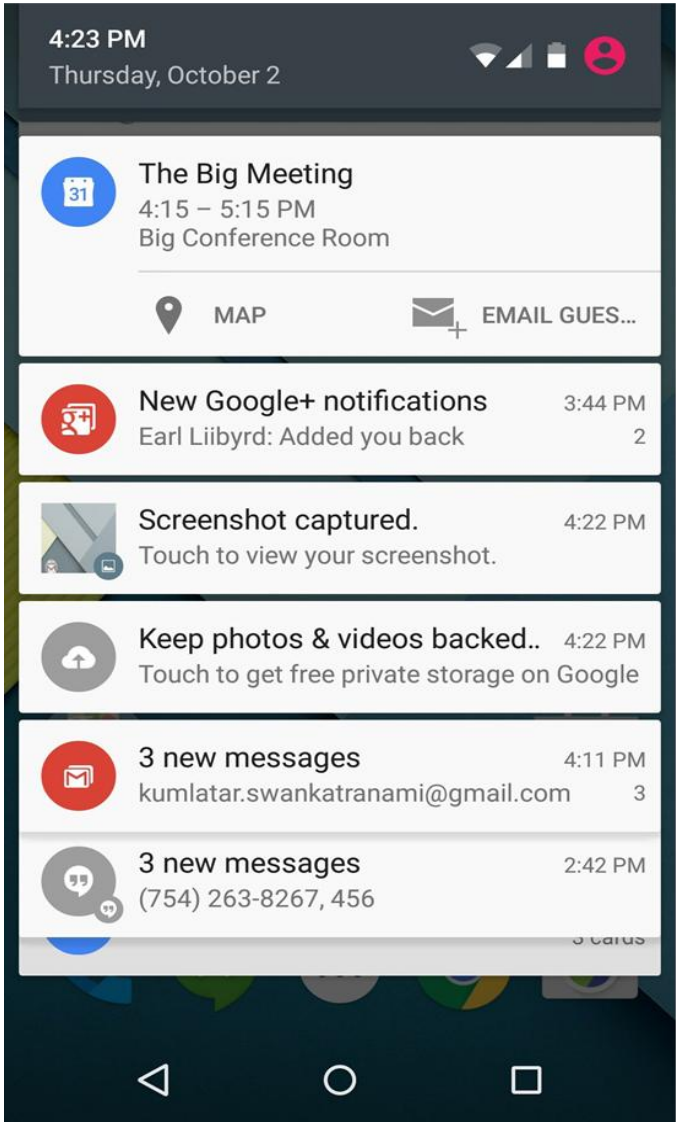
Once the client receives the IBinder , the client interacts with the service through that interface.

The diagram below shows a comparison between the started and bound service lifecycles :

Notifications

What is a notification?

- A notification is a message your app displays to the user outside your application's normal UI.
- When you tell the system to issue a notification, the notification first appears to the user as an icon in the notification area, on the left side of the status bar.
- To see the details of the notification, the user opens the notification drawer, or views the notification on the lock screen if the device is locked.
- The notification area, the lock screen, and the notification drawer are system-controlled areas that the user can view at any time.



Creating notifications :

You create a notification using the NotificationCompat.Builder class. The builder classes simplify the creation of complex objects. To create a NotificationCompat.Builder , pass the application context to the constructor:

Setting notification components :

When using NotificationCompat.Builder, you must assign a small icon, text for a title, and the notification message. You should keep the notification message shorter than 40 characters and not repeat what's in the title. For example:

```
NotificationCompat.Builder mBuilder =
new NotificationCompat.Builder(this)
.setSmallIcon(R.drawable.notification_icon)
.setContentTitle("Dinner is ready!")
.setContentText("Lentil soup, rice pilaf, and cake for dessert.");
```

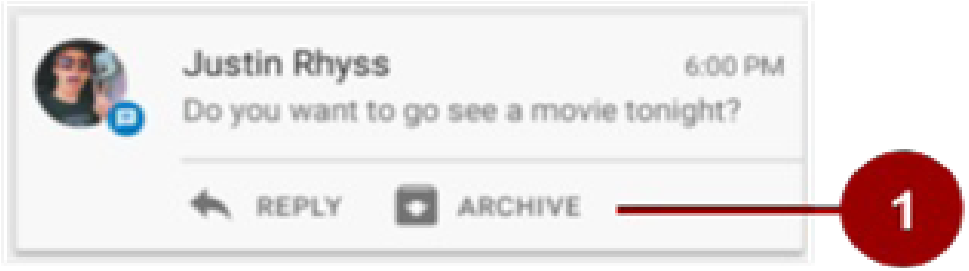
You also need to set an Intent that determines what happens when the user clicks the notification. Usually this Intent results in your app launching an Activity.

You can use various options with notifications, including:

1. Notification actions
2. Priorities
3. Expanded layouts
4. Ongoing notifications

1.Notification actions

- A notification action is an action that the user can take on the notification.
- The action is made available via an action button on the notification.
- Like the Intent that determines what happens when the user clicks the notification, a notification action uses a PendingIntent to complete the action.
- The Android system usually displays a notification action as a button adjacent to the notification content.
- Starting with Android 4.1 (API level 16), notifications support icons embedded below the content text, as shown in the screenshot below



2.Priorities :

- Android allows you to assign a priority level to each notification to influence how the Android system will deliver it.
- Notifications have a priority between MIN (-2) and MAX (2) that corresponds to their importance. The following table
- shows the available priority constants defined in the Notification class.

Priority Constant	Use
PRIORITY_MAX	For critical and urgent notifications that alert the user to a condition that is time-critical or needs to be resolved before they can continue with a time-critical task.
PRIORITY_HIGH	Primarily for important communication, such as messages or chats.
PRIORITY_DEFAULT	For all notifications that don't fall into any of the other priorities described here.
PRIORITY_LOW	For information and events that are valuable or contextually relevant, but aren't urgent or time-critical.
PRIORITY_MIN	For nice-to-know background information. For example, weather or nearby places of interest.

Scheduling Alarms

You already know how to use broadcast receivers to make your app respond to system events even when your app isn't running. In this chapter, you'll learn how to use alarms to schedule tasks for specific times, whether or not your app is running at the time the alarm is set to go off.

Alarms can either be single use or repeating. For example, you can use a repeating alarm to schedule a download every day at the same time.

To create alarms, you use the AlarmManager class. Alarms in Android have the following characteristics:

- Alarms let you send intents at set times or intervals. You can use alarms with broadcast receivers to start services and perform other operations.
- Alarms operate outside your app, so you can use them to trigger events or actions even when your app isn't running, and even if the device is asleep.
- When used correctly, alarms can help you minimize your app's resource requirements. For example, you can schedule operations without relying on timers or continuously running background services.

Alarm types

There are two general types of alarms in Android:

1. Elapsed real-time alarms
2. Real-time clock (RTC) alarms, and both use PendingIntent objects.

1. Elapsed real-time alarms

Elapsed real-time alarms use the time, in milliseconds, since the device was booted. Elapsed real-time alarms aren't affected by time zones, so they work well for alarms based on the passage of time. For example, use an elapsed real-time alarm for an alarm that fires every half hour.

The AlarmManager class provides two types of elapsed real-time alarm:

ELAPSED_REALTIME : Fires a PendingIntent based on the amount of time since the device was booted, but doesn't wake the device. The elapsed time includes any time during which the device was asleep. All repeating alarms fire when your device is next awake.

ELAPSED_REALTIME_WAKEUP : Fires the PendingIntent after the specified length of time has elapsed since device boot, waking the device's CPU if the screen is off. Use this alarm instead of ELAPSED_REALTIME if your app has a time dependency, for example if it has a limited window during which to perform an operation.

2.Real-time clock (RTC) alarms

Real-time clock (RTC) alarms are clock-based alarms that use Coordinated Universal Time (UTC). Only choose an RTC alarm in these types of situations:

- You need your alarm to fire at a particular time of day. The alarm time is dependent on current locale.
- Apps with clock-based alarms might not work well across locales, because they might fire at the wrong times. And if the user changes the device's time setting, it could cause unexpected behavior in your app.
- The AlarmManager class provides two types of RTC alarm:
- RTC : Fires the pending intent at the specified time but doesn't wake up the device. All repeating alarms fire when your device is next awake.
- RTC_WAKEUP : Fires the pending intent at the specified time, waking the device's CPU if the screen is off.

Scheduling an alarm :

The AlarmManager class gives you access to the Android system alarm services. AlarmManager lets you broadcast an Intent at a scheduled time, or after a specific interval.

To schedule an alarm:

1. Call getSystemService(ALARM_SERVICE) to get an instance of the AlarmManager class.

2. Use one of the set…() methods available in AlarmManager (as described below). Which method you use depends on whether the alarm is elapsed real time, or RTC.

All the AlarmManager.set…() methods include these two arguments:

A type argument, which is how you specify the alarm type:

ELAPSED_REALTIME or ELAPSED_REALTIME_WAKEUP , described in Elapsed real-time alarms above. RTC or RTC_WAKEUP , described in Real-time clock (RTC) alarms above. A PendingIntent object, which is how you specify which task to perform at the given time.

Scheduling a single-use alarm

To schedule a single alarm, use one of the following methods on the AlarmManager instance:

1.set() : For devices running API 19+, this method schedules a single, inexactly timed alarm, meaning that the system shifts the alarm to minimize wakeups and battery use. For devices running lower API versions, this method schedules an exactly timed alarm.

2.setWindow() : For devices running API 19+, use this method to set a window of time during which the alarm should be triggered.

3. setExact() : For devices running API 19+, this method triggers the alarm at an exact time. Use this method only for alarms that must be delivered at an exact time, for example an alarm clock that rings at a requested time. Exact alarms reduce the OS's ability to minimize battery use, so don't use them unnecessarily.

Canceling an alarm

To cancel an alarm, use cancel() and pass in the PendingIntent . For example: alarmManager.cancel(alarmIntent);

User-visible alarms ("alarm clocks")

- For API 21+ devices, you can set a user-visible alarm clock by calling setAlarmClock() .
- Apps can retrieve the next user visible alarm clock that's set to go off by calling getNextAlarmClock() .
- Alarms clocks set with setAlarmClock() work even when the device or app is idle (similar to setExactAndAllowWhileIdle()), which gets you as close to an exact wake up call as possible.

References/Bibliography

Text Book :

1.Google developer training android developer fundamentals course