Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

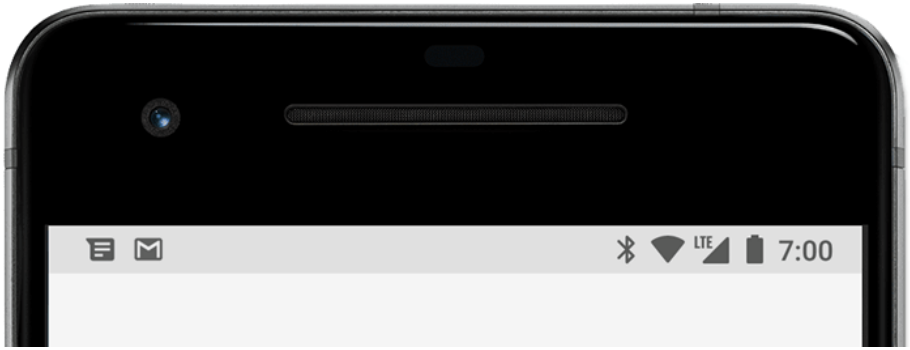
Android Development

Notifications, Permissions, Content Providers

Notifications

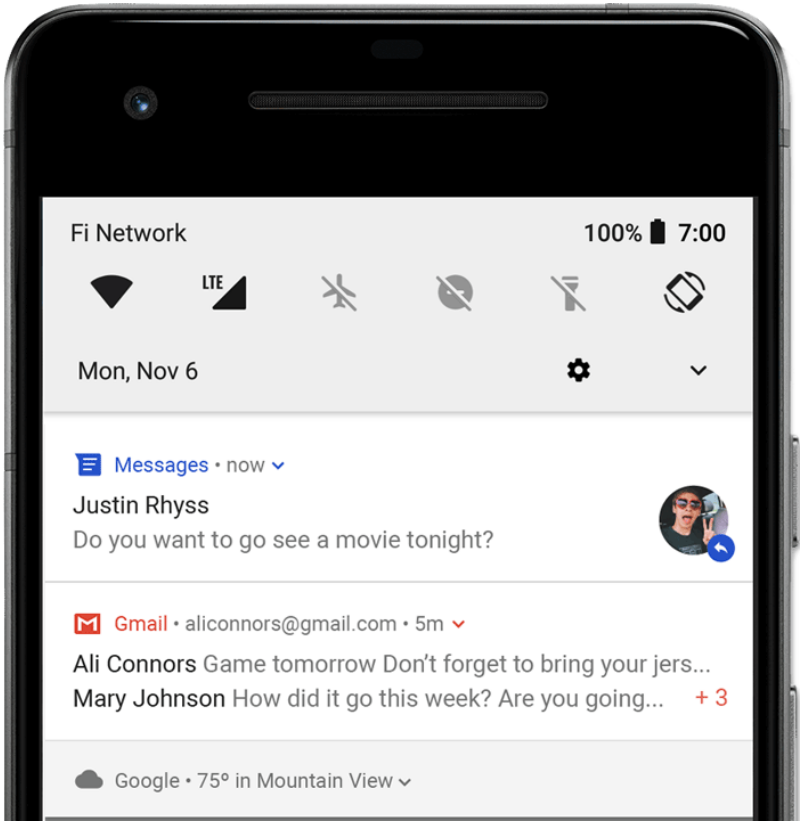
Notifications Overview

- A notification is a message that Android displays outside your app's UI to provide the user with reminders, communication from other people, or other timely information from your app.
- Users can tap the notification to open your app or take an action directly from the notification.

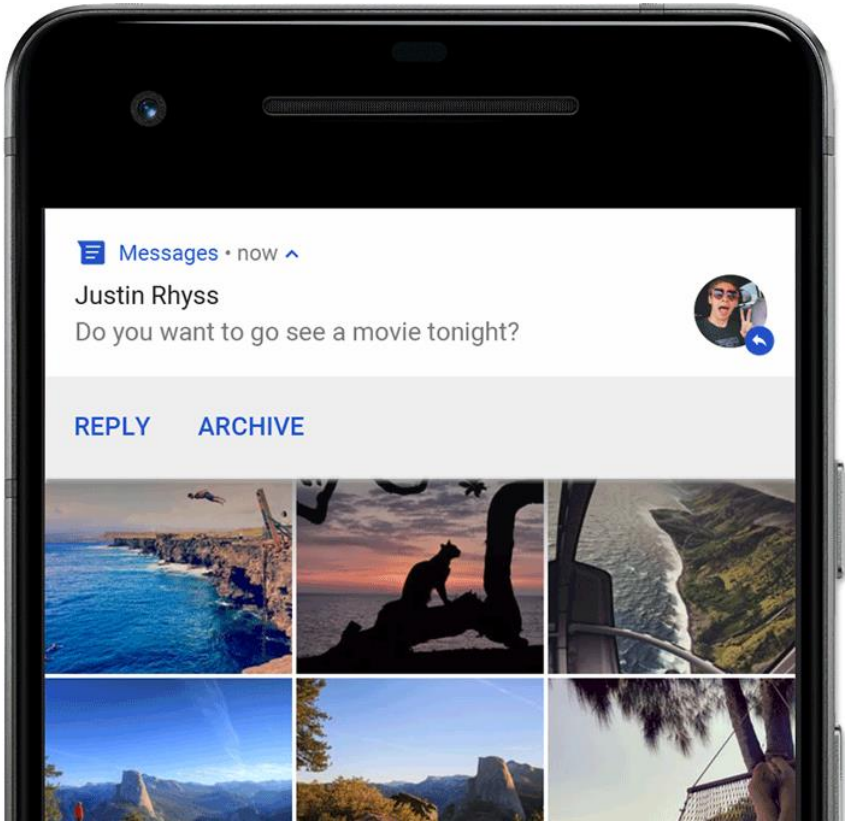


Notifications

Classic notification

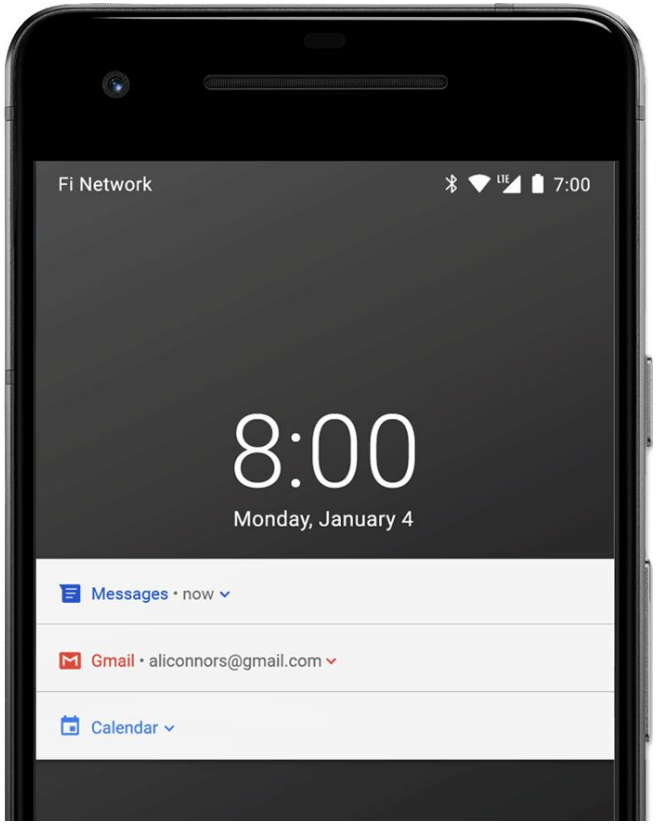


Heads-up notification

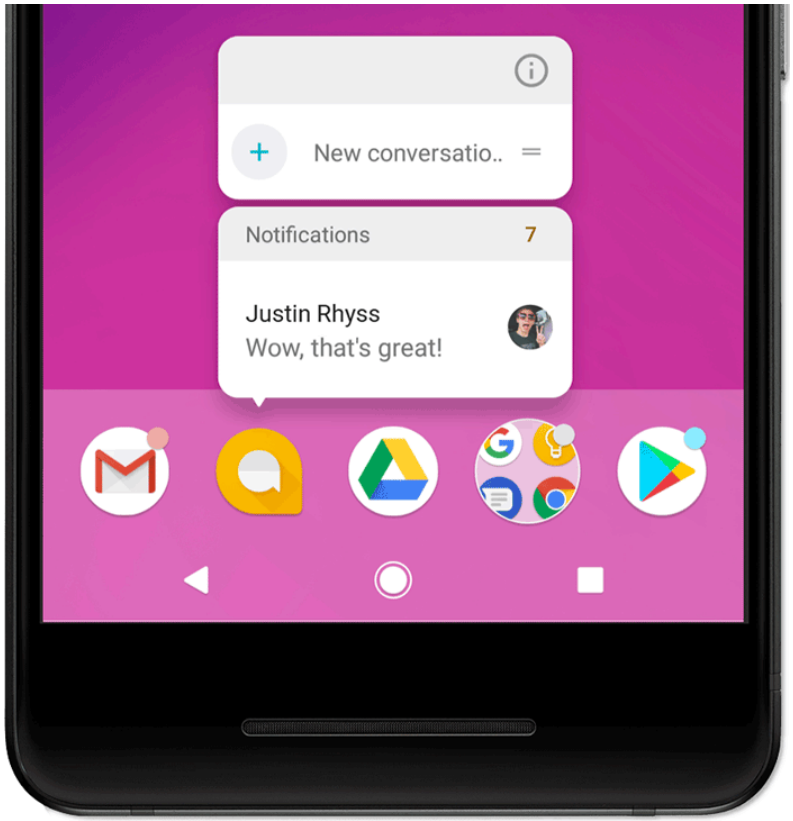


Notifications

Lock screen

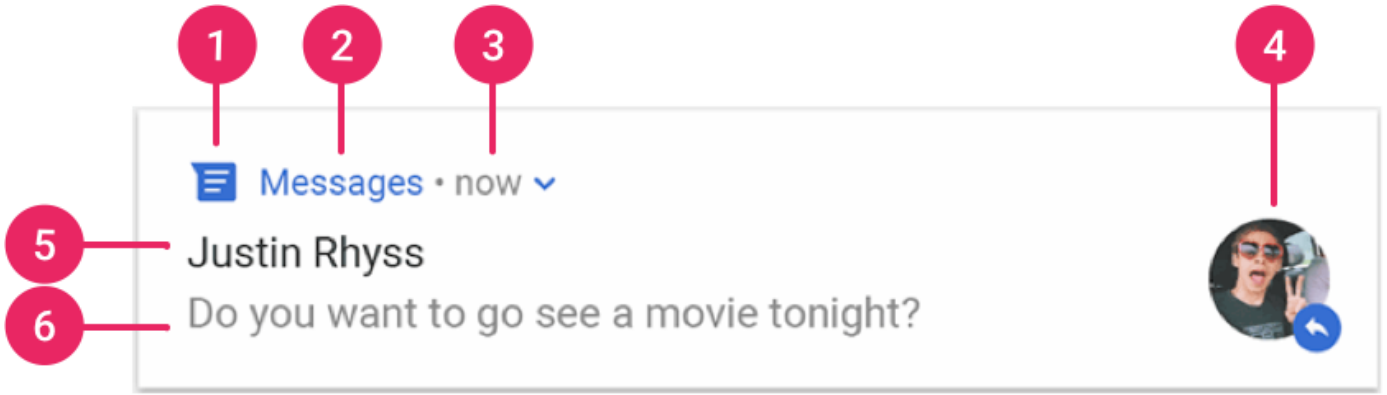


App icon badge



Components

- The design of a notification is determined by system templates—your app simply defines the contents for each portion of the template.
- Some details of the notification appear only in the expanded view.



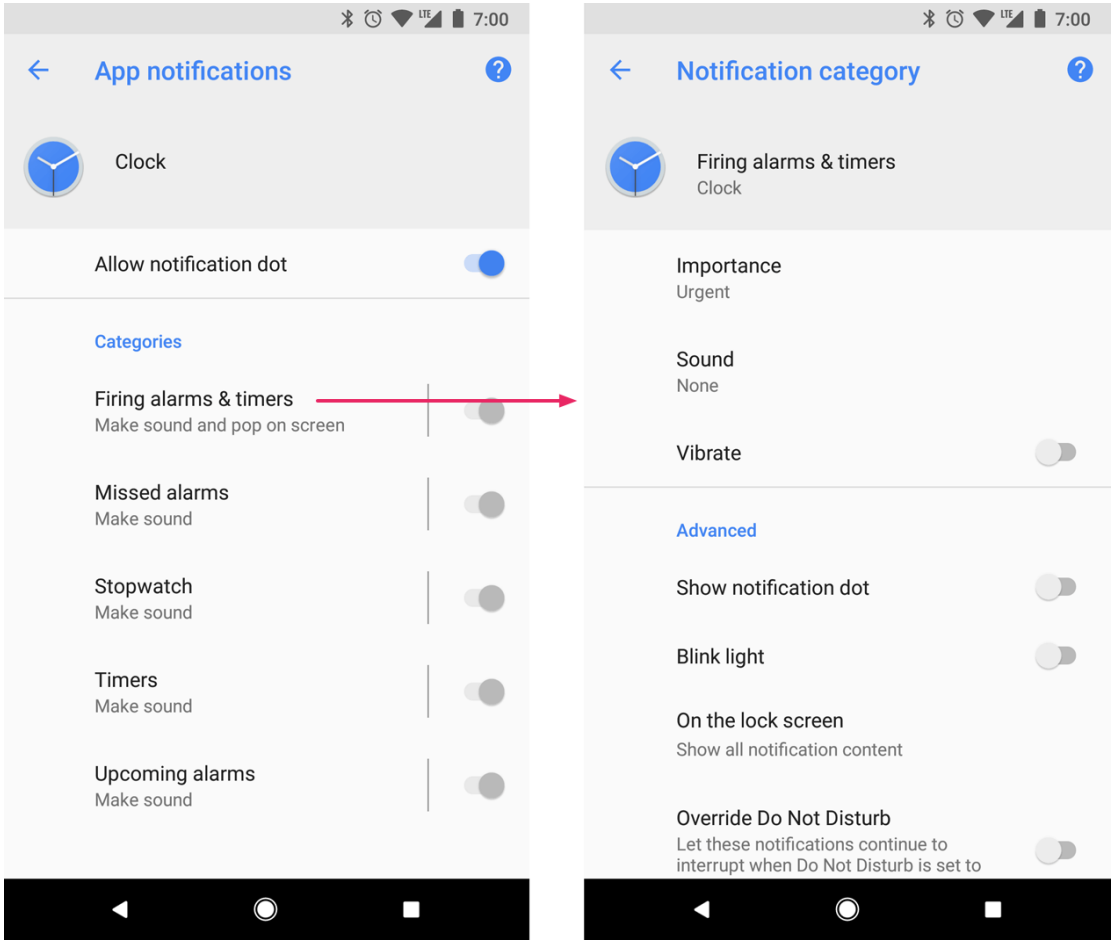
Components

1. Small icon: This is required and set with setSmallIcon().
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with setWhen() or hide it with setShowWhen(false).
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set with setLargeIcon().
5. Title: This is optional and set with setContentTitle().
6. Text: This is optional and set with setContentText().

Notification

- Notification actions
  - Although it's not required, every notification should open an appropriate app activity when tapped.
  - In addition to this default notification action, you can add action buttons that complete an app-related task from the notification
- Expandable notification
  - By default, the notification's text content is truncated to fit on one line.
  - If you want your notification to be longer, you can enable a larger text area that's expandable by applying an additional template

Notification channels



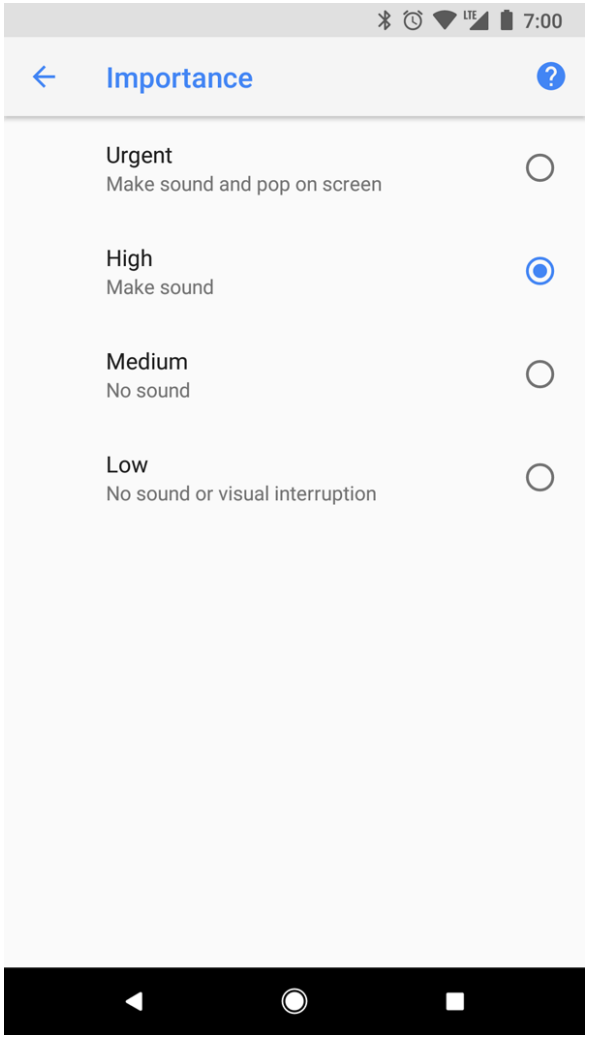
You can set different settings for each notification channel

- On/Off
- Importance
- Show on the lock screen
- Show notification dot

In the app, you can also define custom settings for your notifications.

Notification importance

- Android uses the *importance* of a notification to determine how much the notification should interrupt the user (visually and audibly).
- The higher the importance of a notification, the more interruptive the notification will be.



Notifications for foreground services

- A notification is required when your app is running a "foreground service„
  - This notification cannot be dismissed like other notifications.
  - To remove the notification, the service must be either stopped or removed from the "foreground" state.

Creating notification

- Using builders
  - ```
    var builder = NotificationCompat.Builder(this, CHANNEL_ID)
            .setSmallIcon(R.drawable.notification_icon)
            .setContentTitle(textTitle)
            .setContentText(textContent)
            .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    ```
- Functions
  - A small icon, set by setSmallIcon(). This is the only user-visible content that's required.
  - A title, set by setContentTitle().
  - The body text, set by setContentText().
  - The notification priority, set by setPriority().
    - For Android 8.0 and higher, you must instead set the channel importance
    ```

Creating notification

- By default, the notification's text content is truncated to fit one line.
  - If you want your notification to be longer, you can enable an expandable notification by adding a style template with setStyle().
  - For example, the following code creates a larger text area

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Much longer text that cannot fit one line...")
        .setStyle(NotificationCompat.BigTextStyle()
                .bigText("Much longer text that cannot fit one line..."))
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

Notification Channel

- Before you can deliver the notification on Android 8.0 and higher, you must register your app's notification channel with the system by passing an instance of NotificationChannel to createNotificationChannel().
- So the following code is blocked by a condition on the SDK_INT version:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    val name = getString(R.string.channel_name)
    val descriptionText = getString(R.string.channel_description)
    val importance = NotificationManager.IMPORTANCE_DEFAULT
    val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
        description = descriptionText
    }
    // Register the channel with the system
    val notificationManager: NotificationManager =
        getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
    notificationManager.createNotificationChannel(channel)
}
```

Showing

```
with(NotificationManagerCompat.from(this)) {
    // notificationId is a unique int for each
    //       notification that you must define
    notify(notificationId, builder.build())
}
```

Notification customisation

- Big notification text
  - `.setStyle(new NotificationCompat.BigTextStyle()`
    `.bigText("Much longer text that cannot fit one line..."))`
- Notification light
  - `.setLights(Color.RED,100,100)`
- Vibration
  - `.setVibrate(new long[]{123,123,123,123})`
- Notification icon
  - `.setSmallIcon(R.drawable.notification_icon)`
  - `.setLargeIcon(Bitmap)`
- Custom Action
  - `.addAction(R.drawable.icon, "Call", pIntent)`
  - The custom action can be a button press or a direct text input
- Progress Bar
  - `mBuilder.setProgress(PROGRESS_MAX, PROGRESS_CURRENT, false);`
    `notificationManager.notify(notificationId, mBuilder.build());`

Notification badge

- Starting with 8.0 (API level 26), notification badges (also known as notification dots) appear on a launcher icon when the associated app has an active notification.
  - Users can long-press on the app icon to reveal the notifications (alongside any app shortcuts),
  - These dots appear by default in launcher apps that support them and there's nothing your app needs to do.
  - However, there might be situations in which you don't want the to notification dot to appear or you want to control exactly which notifications to appear there.

Notification badge

- Disable badging

```
val id = "my_channel_01"
val name = getString(R.string.channel_name)
val descriptionText = getString(R.string.channel_description)
val importance = NotificationManager.IMPORTANCE_LOW
val mChannel = NotificationChannel(id, name, importance).apply {
    description = descriptionText
    setShowBadge(false)
}
val notificationManager =
getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
notificationManager.createNotificationChannel(mChannel)
```

Vibrations and LED

- The Notification builder has several additional functions.
  - `setVibrate(long[] pattern)`
    - The pattern describes the pauses and active time periods
- Through the notification channel
  - `public void enableLights (boolean lights)`
    - LED
  - `public void enableVibration (boolean vibration)`
  - `public void setVibrationPattern (long[] vibrationPattern)`
  - `public void setSound (Uri sound, AudioAttributes audioAttributes)`

Permissions

Permissions

- Permission are added to the Android manifest file:

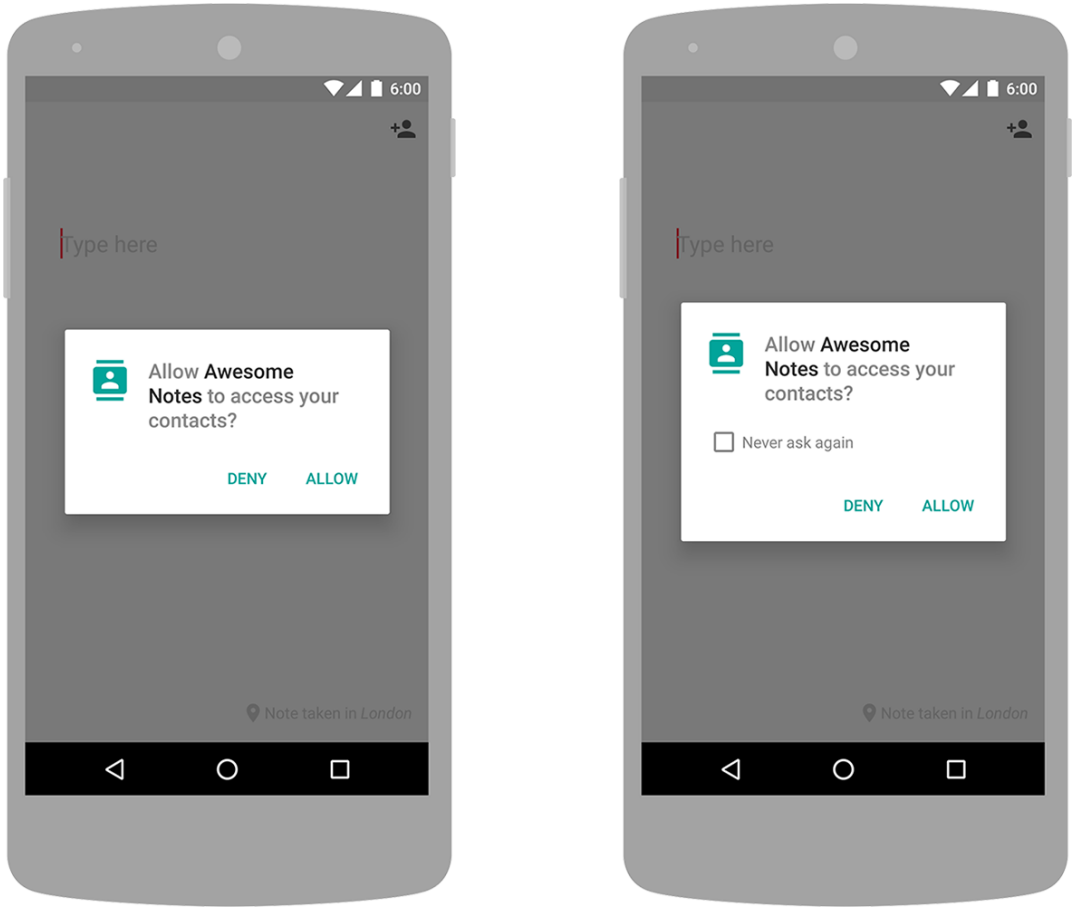
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="lecture_example.com.myapplication">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

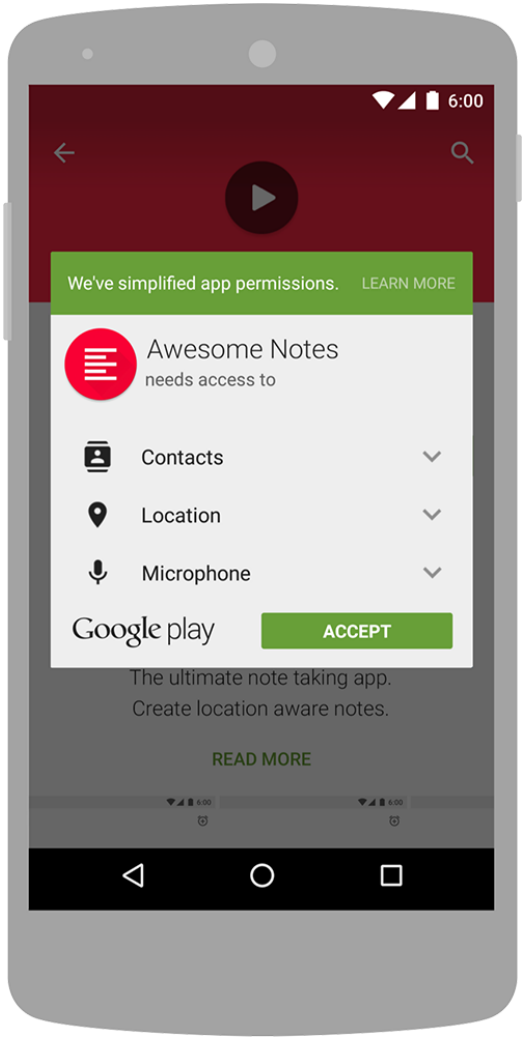
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```


6:00
Type here
Allow Awesome Notes to access your contacts?
DENY
ALLOW
Note taken in London
6:00
Type here
Allow Awesome Notes to access your contacts?
Never ask again
DENY
ALLOW
Note taken in London

Runtime Permissions

- In the pre 6.0 Android days, we only had to add our permissions in the manifest file, and they worked like a charm.
- Since 6.0 we also have to handle the permissions when the app is up and running hence the name runtime permissions.



Runtime Permissions

- First we need to check if a certain permission is already requested:
  - `ContextCompat.checkSelfPermission(thisActivity, Manifest.permission.WRITE_CALENDAR)`
- The result of this check can be:
  - PackageManager.PERMISSION_GRANTED
  - PackageManager.PERMISSION_DENIED
- The syntax of a permission request:
  - ```
    ActivityCompat.requestPermissions(thisActivity,
                      arrayOf(Manifest.permission.READ_CONTACTS),
                      MY_PERMISSIONS_REQUEST_READ_CONTACTS)
    ```
- Note the permission request has three inputs:
  - Context
  - Array of permissions
  - Request id
    ```

Runtime Permissions

When you need runtime permissions:

- READ_CALENDAR
- WRITE_CALENDAR
- CAMERA
- READ_CONTACTS
- WRITE_CONTACTS
- GET_ACCOUNTS
- ACCESS_FINE_LOCATION
- ACCESS_COARSE_LOCATION
- RECORD_AUDIO
- READ_PHONE_STATE
- READ_PHONE_NUMBERS
- CALL_PHONE
- ANSWER_PHONE_CALLS
- READ_CALL_LOG
- WRITE_CALL_LOG

- READ_CALENDAR
- ADD_VOICEMAIL
- USE_SIP
- PROCESS_OUTGOING_CALLS
- BODY_SENSORS
- SEND_SMS
- RECEIVE_SMS
- READ_SMS
- RECEIVE_WAP_PUSH
- RECEIVE_MMS
- READ_EXTERNAL_STORAGE
- WRITE_EXTERNAL_STORAGE

Runtime Permissions

- Catching the response from the permission request:

```
override fun onRequestPermissionsResult(requestCode: Int,
                        permissions: Array<String>, grantResults: IntArray) {
    when (requestCode) {
        MY_PERMISSIONS_REQUEST_READ_CONTACTS -> {

            if ((grantResults.isNotEmpty() && grantResults[0] ==
                                        PackageManager.PERMISSION_GRANTED)) {

            } else {

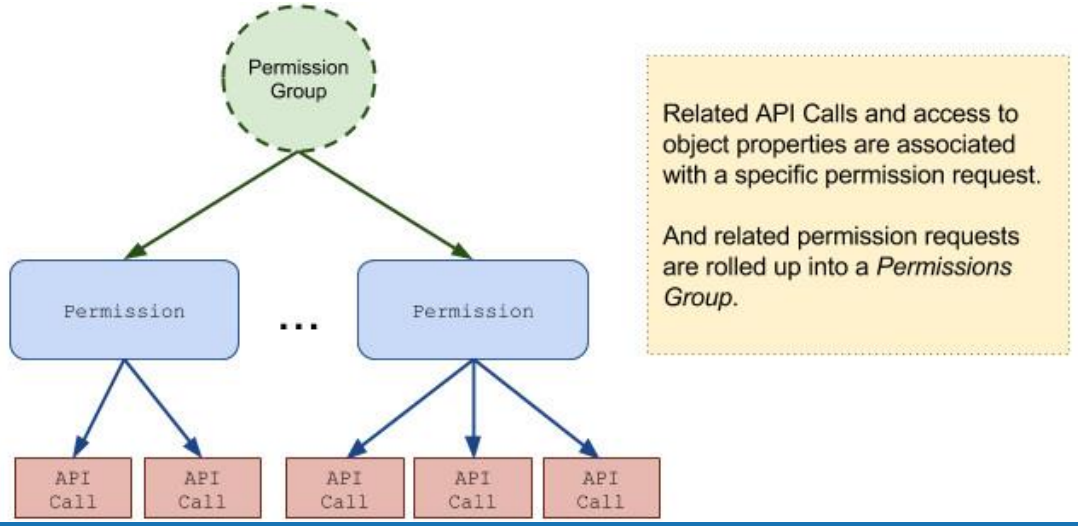
            }
            return
        }
        else -> {

        }
    }
}
```

- Normal and Dangerous Permissions

Permission groups

- Permissions are organized into groups related to a device's capabilities or features.
  - Under this system, permission requests are handled at the group level and a single permission group corresponds to several permission declarations in the app manifest.



Storage

Shared Preferences

- It stores the primitive data in key – value pairs:
  - boolean, float, int, long, string, stringSet
- Data stored in shared preferences are kept safe after the app is closed.
- For the settings of an app, it is a good practice to use: PreferenceActivity
- When we use the Shared Preferences we need an Editor:
  - `SharedPreferences.Editor editor = settings.edit();`

Shared Preferences

- An example for shared preferences usage:
  - ```
    val sharedPref = activity?.getSharedPreferences(
            getString(R.string.preference_file_key),
                                                Context.MODE_PRIVATE)
    ```
  - ```
    val sharedPref =
    activity?.getPreferences(Context.MODE_PRIVATE) ?: return
    with (sharedPref.edit()) {
        putInt(getString(R.string.saved_high_score_key),
                                                        newHighScore)
        commit()
    }
    ```

SQLite

- Accessed by name
- Private to the application
- You may use the SQLiteOpenHelper class and override its onCreate() method

SQLite

- Example:

```
private const val SQL_CREATE_ENTRIES =
        "CREATE TABLE ${FeedEntry.TABLE_NAME} (" +
                "${BaseColumns._ID} INTEGER PRIMARY KEY," +
                "${FeedEntry.COLUMN_NAME_TITLE} TEXT," +
                "${FeedEntry.COLUMN_NAME_SUBTITLE} TEXT)"

private const val SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS
${FeedEntry.TABLE_NAME}"
```

SQLite

- Example:
- ```
  class FeedReaderDbHelper(context: Context) : SQLiteOpenHelper(context,
  DATABASE_NAME, null, DATABASE_VERSION) {
      override fun onCreate(db: SQLiteDatabase) {
          db.execSQL(SQL_CREATE_ENTRIES)
      }
      override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
                                                 newVersion: Int) {
          db.execSQL(SQL_DELETE_ENTRIES)
          onCreate(db)
      }
      override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int,
                                                   newVersion: Int) {
          onUpgrade(db, oldVersion, newVersion)
      }
      companion object {

          const val DATABASE_VERSION = 1
          const val DATABASE_NAME = "FeedReader.db"
      }
  }
  ```
  ```

SQLite

- Example:

```
// Gets the data repository in write mode
val db = dbHelper.writableDatabase

// Create a new map of values, where column names are the keys
val values = ContentValues().apply {
    put(FeedEntry.COLUMN_NAME_TITLE, title)
    put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle)
}

// Insert the new row, returning the primary key value of the new row
val newRowId = db?.insert(FeedEntry.TABLE_NAME, null, values)
```

SQLite – Query

```
val db = dbHelper.readableDatabase

// Define a projection that specifies which columns from the database
// you will actually use after this query.
val projection = arrayOf(BaseColumns._ID, FeedEntry.COLUMN_NAME_TITLE,
FeedEntry.COLUMN_NAME_SUBTITLE)

// Filter results WHERE "title" = 'My Title'
val selection = "${FeedEntry.COLUMN_NAME_TITLE} = ?"
val selectionArgs = arrayOf("My Title")

// How you want the results sorted in the resulting Cursor
val sortOrder = "${FeedEntry.COLUMN_NAME_SUBTITLE} DESC"

val cursor = db.query(
        FeedEntry.TABLE_NAME,   // The table to query
        projection,             // The array of columns to return (pass null to get all)
        selection,              // The columns for the WHERE clause
        selectionArgs,          // The values for the WHERE clause
        null,                   // don't group the rows
        null,                   // don't filter by row groups
        sortOrder               // The sort order
)
```

SQLite – Query

- ```
  val itemIds = mutableListOf<Long>()
  with(cursor) {
      while (moveToNext()) {
          val itemId =
  getLong(getColumnIndexOrThrow(BaseColumns._ID))
          itemIds.add(itemId)
      }
  }
  ```
  ```

File Storage

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, getFilesDir() or getCacheDir() From external storage, getExternalFilesDir() or getExternalCacheDir()	Never needed for internal storage Not needed for external storage when your app is used on devices that run Android 4.4 (API level 19) or higher	No, if files are in a directory within internal storage Yes, if files are in a directory within external storage	Yes
Media	Shareable media files (images, audio files, videos)	MediaStore API	READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE when accessing other apps' files on Android 10 (API level 29) or higher Permissions are required for all files on Android 9 (API level 28) or lower	Yes, though the other app needs the READ_EXTERNAL_STORAGE permission	No
Documents and other files	Other types of shareable content, including downloaded files	Storage Access Framework	None	Yes, through the system file picker	No
App preferences	Key-value pairs	Jetpack Preferences library	None	No	Yes
Database	Structured data	Room persistence library	None	No	Yes

Internal Storage

- Private data
  - After uninstall, the data is deleted
    - only in Android versions prior 6.0
  - Example:
  - ```
    val filename = "myfile"
    val fileContents = "Hello world!"
    context.openFileOutput(filename,
    Context.MODE_PRIVATE).use {
            it.write(fileContents.toByteArray())
    }
    ```
    ```

Internal Storage

- `getFilesDir()`
  - The absolute path to the storage of the application
- `getDir()`
  - Create and/or open a directory
- `deleteFile()`
- `fileList()`
  - Returns an array with the file list

External Storage

- Public storage
  - Can be the SD card as well the internal memory (public storage)
  - There is no security access
  - Example:

```
fun isExternalStorageWritable(): Boolean {
    return Environment.getExternalStorageState() ==
Environment.MEDIA_MOUNTED
}

fun isExternalStorageReadable(): Boolean {
     return Environment.getExternalStorageState() in
        setOf(Environment.MEDIA_MOUNTED,
Environment.MEDIA_MOUNTED_READ_ONLY)
}
```

External Storage

- Further features
  - `getExternalFilesDir()`
    - Returns the absolute path to the directory on the primary shared/external storage device where the application can place persistent files it owns.
  - `getExternalStoragePublicDirectory()`
    - There are several pre-defined categories
      - Music, Podcasts, Ringtones, Alarms, Notifications, Pictures, Movies, Download
  - `getCacheDir()`
    - Returns the absolute path to the application specific cache directory on the filesystem.
  - `getExternalCacheDir()`
    - Returns absolute path to application-specific directory on the primary shared/external storage device where the application can place cache files it owns.
  - `getExternalStorageDirectory()`
    - Return the primary shared/external storage directory.

Temporary files

- You can use temporary files:

```
File.createTempFile(filename, null, context.cacheDir)
```

Firebase

- „Firebase handles the backend online element for your apps, allowing you to focus on the front-end UI and functionality."
- „Firebase removes the need to create your own server-side script using PHP and MySQL, or a similar set-up.
  - This is 'Backend as a Service' or 'BaaS', and essentially this means that anyone really *can* make that ambitious social app."
- Setting up a project
  - Before you can do anything with Firebase, you first need to create an account.
    - You can do this over at firebase.google.com.
  - On the console page you can add new projects to your Firebase, but Android Studio is now able to accomplish this.
    - Tools -> Firebase : this will open Firebase Assistant from where you can add Firebase features, for example Realtime Database.
    - Once you open the wizard it will guide you step-by-step how to add a Realtime Database

Firebase

- Creating the database manager:
  - A good approach to the firebase database manager is to use a singleton class.
  - In the singleton class you will need to add the following to access database features:
    - `val database = FirebaseDatabase.getInstance()`
  - To get references to the database nodes the syntax is:
    - `val myRef = database.getReference("message")`
  - To write data to your database you have multiple options:
    - Direct type write this will overwrite existing data:
      - `myRef.setValue("Hello, World!")`
    - Update type write:
      - `myRef.updateChildren(Map<String,Object>);`

Firebase

- Read from database:
  - Single Value:
    - ```
      myRef.addValueEventListener(object : ValueEventListener {
          override fun onDataChange(dataSnapshot: DataSnapshot) {
          // This method is called once with the initial value and again
          // whenever data at this location is updated.
              val value = dataSnapshot.getValue(String::class.java)
              Log.d(TAG, "Value is: $value")
          }

          override fun onCancelled(error: DatabaseError) {
              // Failed to read value
              Log.w(TAG, "Failed to read value.", error.toException())
          }
      })
      ```
  - Single Value when changed:
    - `myRef.addValueEventListener(new ValueEventListener(){…}`
      ```

Firebase

- Firebase accepts:
  - `String`
  - `Long`
  - `Double`
  - `Boolean`
  - `Map<String, Object>`
  - `List<Object>`
- If you use a Java object, the contents of your object are automatically mapped to child locations in a nested fashion.
- Firebase Realtime Database is structured like a big JSON file.

Firebase

- Add data to the database in more detail:
  - Structured data

```
{
  "users": {
    "alovelace": {
      "name": "Ada Lovelace",
      "email": "ada@prog.org"},
    },
    "ghopper": { ... },
    "eclarke": { ... }
  }
}
```

Firebase

- Class for data
  - ```
    @IgnoreExtraProperties
    data class User(
        var username: String? = "",
        var email: String? = ""
    )
    ```
    ```

Firebase

- Add data to the database in more detail:
- We want to add a new user:

```
private fun writeNewUser(userId: String, name: String, email: String?) {
    val user = User(name, email)
    database.child("users").child(userId).setValue(user)
}
```

- We want to modify part of a user:

```
database.child("users").child(userId).child("username").setValue(name)
```

Firebase

- Update multiple fields in a single request:
  - In the updateChildren method you need to pass a map which is containing the fields you want to update:

Example

```
@IgnoreExtraProperties
data class Post(
    var uid: String? = "",
    var author: String? = "",
    var title: String? = "",
    var body: String? = "",
    var starCount: Int = 0,
    var stars: MutableMap<String, Boolean> = HashMap()
) {

    @Exclude
    fun toMap(): Map<String, Any?> {
        return mapOf(
                "uid" to uid,
                "author" to author,
                "title" to title,
                "body" to body,
                "starCount" to starCount,
                "stars" to stars
        )
    }
}
```

Firebase

- You can add to every Firebase call a CompletionListener(). The completion listener will inform you that the write procedure was successful or it failed.
  - The CompletionListener has a return function:
    - ```
      @Override
      public void onComplete(final DatabaseError
      databaseError, DatabaseReference databaseReference)
      {}
      ```
    - If the database error is null then the task was successful, if it failed this object will contain the detail about the error.
      ```

Firebase

- Listening for events
  - A firebase instance can listen for certain database changes:
    - onChildAdded() – triggers when a data is added to a certain node
    - onChildChanged() – triggers when data is changed in a certain node
    - onChildRemoved() – triggers when data is removed in a certain node
    - onChildMoved() – triggers when a data is moved (order changed) from its location in a certain node
    - onCancelled() – triggers when the request failed
  - You can add a child event listener to every Firebase reference:
    - `ref.addChildEventListener(childEventListener);`
  - NOTE !! If your application is listening to a certain node and a second instance of your app is writing data into this node, all instance listening will be triggered.
  - The listening will be continuous until you cancel it by removing the listener from the specific database reference.
  - More info on list of data

Firebase

- Rules
  - Rules in Firebase Realtime Database determine who can access certain nodes, and also you can determine the validity of certain nodes.
  - In our apps we will use the simplest rules:
  - ```
    "rules": {
      ".read": "true",
      ".write": "true"}
    ```
  - More info on rules
    ```

Firebase

- Firebase comes packed with offline capabilities
  - You have the option to use the Firebase Realtime database when you are offline
  - In this way, the Firebase instance will work on a cached copy of the database, and when you are online again, it will sync its data with the server.
  - To use this feature:
    - FirebaseDatabase.getInstance().setPersistenceEnabled(true);
  - If you want to keep a specific node synced in your cached data:
    - ref.keepSynced(true);
  - More info on offline capabilities

Homework

- Create an Android application with two Activities:
  - In the first Activity you should display a user data input form with fields for username, description, email, and a send button.
    - If you press the send button the data should be saved in a firebase database under a users node (the user object which you will send sound include also a user id field for convenience but this should be auto-generated)
    - In conjunction with the firebase save you should open the second activity.
    - Hint: when handling button press do the actions only when the database write was successful, use CompletionHandlers.
  - In the second Activity you should implement a RecyclerView which contains all the users from firebase.
    - The list should display each user in a separate list element. The design of the list element is up on you. The only requirement is that the users should be distinguishable.
  - Please persist the data entered on the first activity:
    - If you change the orientation of the phone please persist the entered data. Use shared preferences.
  - Use Observer Patterns

More on UI

Next week