2017 Android of WMNTAA

Communication, Local Storage and File Access

Communication

- Handler
- Broadcast

Handler

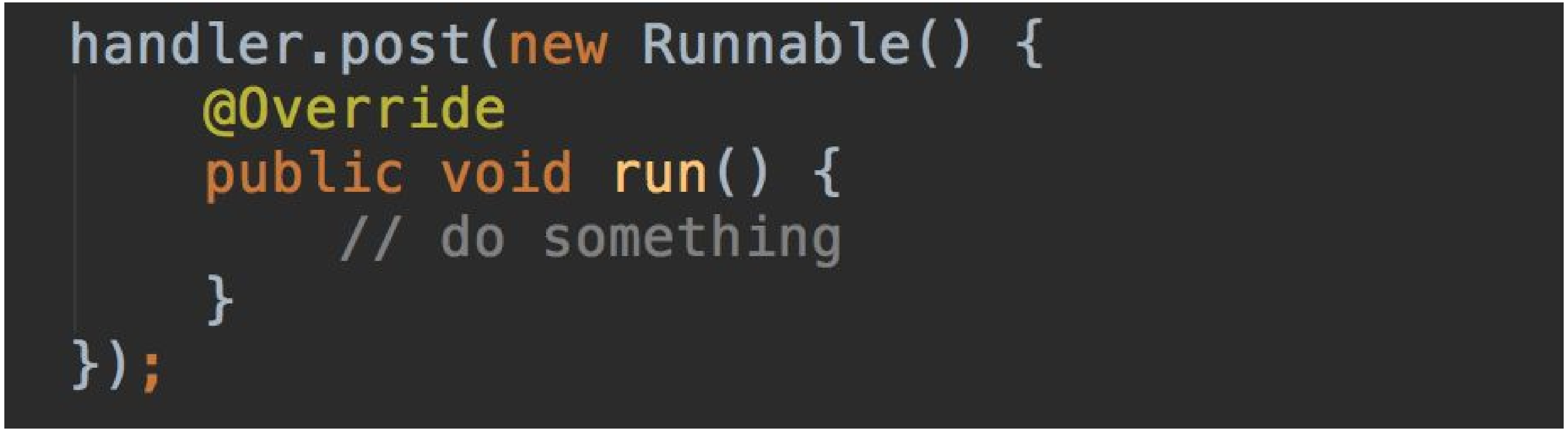
- Overview
- Runnable Task
- Message

Overview

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to the thread / message queue of the thread that is creating it -- from that point on, it will deliver messages and runnables to that message queue and execute them as they come out of the message queue.

Runnable Task

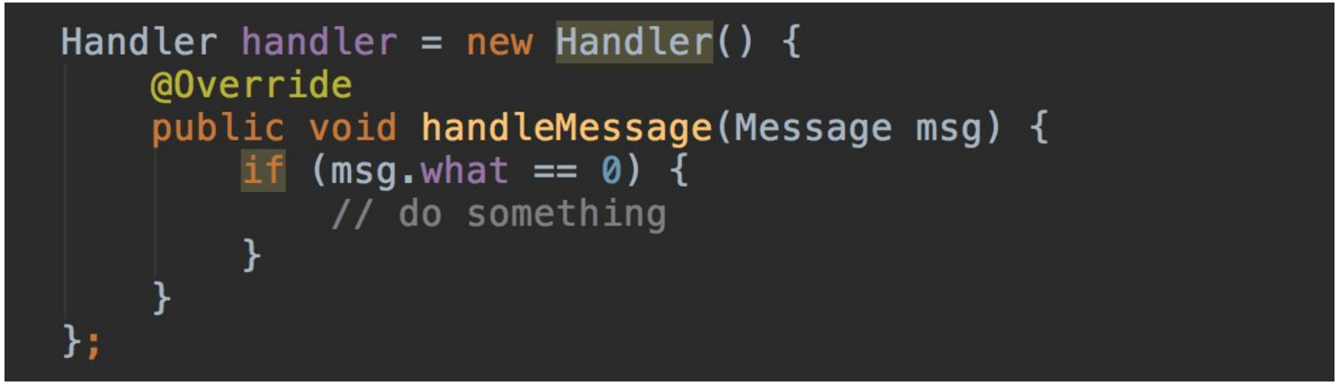
- Scheduling tasks is accomplished with the post(Runnable), postAtTime(Runnable, long), postDelayed(Runnable, long)



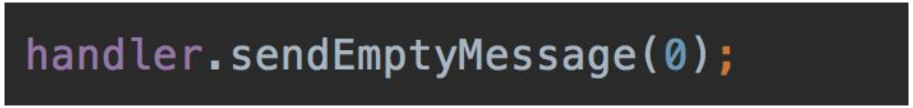
```
handler.post(new Runnable() {
    @Override
    public void run() {
        // do something
    }
});
```

Message

- Scheduling messages is accomplished with the sendEmptyMessage(int), sendMessage(Message), sendMessageAtTime(Message, long), and sendMessageDelayed(Message, long)
- Override Handler.handleMessage(Message) to process Message objects.



```
Handler handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        if (msg.what == 0) {
            // do something
        }
    }
};
```



```
handler.sendEmptyMessage(0);
```

Broadcast

- Intent
- Local Broadcast

Intent

- An intent is an abstract description of an operation to be performed.
- An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed.

Intent

- The primary pieces of information in an intent are:
    - **action** -- The general action to be performed, such as ACTION_VIEW, ACTION_EDIT, ACTION_MAIN, etc.
    - **data** -- The data to operate on, such as a person record in the contacts database, expressed as a Uri.
        - ex : ACTION_VIEW **content://contacts/people/1** -- Display information about the person whose identifier is "1".

Local Broadcast

- You know that the data you are broadcasting won't leave your app, so don't need to worry about leaking private data.
- It is not possible for other applications to send these broadcasts to your app, so you don't need to worry about having security holes they can exploit.
- It is more efficient than sending a global broadcast through the system.

Local Broadcast

- Register

```
LocalBroadcastManager.getInstance(this).registerReceiver(receiver, filter);
```

- Send

```
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

Local Storage and File Access

- Shared Preference
- Internal Storage
- External Storage
- SQLite Database

Shared Preference

- Overview
- Get a Handle to a SharedPreferences
- Write to Shared Preferences
- Read from Shared Preferences

Overview

- A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them. Each SharedPreferences file is managed by the framework and can be private or shared.

Get a Handle to a SharedPreferences

You can create a new shared preference file or access an existing one by calling one of two methods:

- getSharedPreferences() — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any Context in your app.
- getPreferences() — Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

Get a Handle to a SharedPreferences

- getSharedPreferences

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

- getPreferences

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

Write to Shared Preferences

- To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences.
- Pass the keys and values you want to write with methods such as putInt() and putString(). Then call commit() to save the changes.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

Read from Shared Preferences

- To retrieve values from a shared preferences file, call methods such as getInt() and getString(), providing the key for the value you want, and optionally a default value to return if the key isn't present.

```java
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.string.saved_high_score_default);
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), defaultValue);
```

Internal storage

- Overview
- Save a File on Internal Storage

Overview

- It's always available.
- Files saved here are accessible by only your app.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Save a File on Internal Storage

- getFilesDir()
  - Returns a File representing an internal directory for your app.
- getCacheDir()
  - Returns a File representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

Save a File on Internal Storage

- Create a new file

```
File file = new File(context.getFilesDir(), filename);
```

- Cache a file

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

External Storage

- Overview
- Obtain Permissions for External Storage
- Save a File on External Storage

Overview

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from getExternalFilesDir().

External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Obtain Permissions for External Storage

- To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file.

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

Save a File on External Storage

- Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it.
- You can query the external storage state by calling getExternalStorageState(). If the returned state is equal to MEDIA_MOUNTED, then you can read and write your files.

Save a File on External Storage

```java
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Save a File on External Storage

- Public files
  - Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.
  - For example, photos captured by your app or other downloaded files.
- Private files
  - Files that rightfully belong to your app and should be deleted when the user uninstalls your app. When the user uninstalls your app, the system deletes all files in your app's external private directory.
  - For example, additional resources downloaded by your app or temporary media files.

Save a File on External Storage

- Example of public files

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Save a File on External Storage

- Example of private files

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

SQLite Database

- Overview
- Define a Schema and Contract
- Create a Database Using a SQL Helper
- Put Information into a Database
- Read Information from a Database

Overview

- Saving data to a database is ideal for repeating or structured data, such as contact information.

Define a Schema and Contract

- One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized.

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract class,
    // make the constructor private.
    private FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    }
}
```

Create a Database Using a SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table.

```java
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_TITLE + " TEXT," +
    FeedEntry.COLUMN_NAME_SUBTITLE + " TEXT)";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Create a Database Using a SQL Helper

A useful set of APIs is available in the SQLiteOpenHelper class.

```java
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

Create a Database Using a SQL Helper

To access your database, instantiate your subclass of SQLiteOpenHelper.

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

Put Information into a Database

Insert data into the database by passing a ContentValues object to the insert() method

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

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

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

Read Information from a Database

To read from a database, use the query() method, passing it your selection criteria and desired columns.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_SUBTITLE
    };

// Filter results WHERE "title" = 'My Title'
String selection = FeedEntry.COLUMN_NAME_TITLE + " = ?";
String[] selectionArgs = { "My Title" };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_SUBTITLE + " DESC";

Cursor cursor = db.query(
    FeedEntry.TABLE_NAME,                     // The table to query
    projection,                               // The columns to return
    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
    );
```

Challenge

- Create an app for text files writing and reading.
- Two functions :
  - Create and write text files
  - List text files on UI to read by clicking
- How to store text files :
  - Use database with three columns, ID, NAME, and PATH, to store file information
  - Store text files in internal or external storage
- UI thread is only to update UI. So you should do IO in background.