UNIT IV

Storage

- Saving and loading data is essential for most applications.
- At a minimum, an Activity should save its user interface (UI) state before it becomes inactive to ensure the same UI is presented when it restarts.
- It’s also likely that you’ll need to save user preferences and UI selections.

- Android’s nondeterministic Activity and application lifetimes make persisting UI state and application data between sessions particularly important, as your application process may have been killed and restarted before it returns to the foreground.
- Android offers several alternatives for saving application data, each optimized to fulfill a particular need.

- Shared Preferences are a simple, lightweight name/value pair (NVP) mechanism for saving primitive application data, most commonly a user’s application preferences.

- Android also offers a mechanism for recording application state within the Activity lifecycle handlers, as well as for providing access to the local filesystem, through both specialized methods and the java.io classes

- SAVING SIMPLE APPLICATION DATA
  - The data-persistence techniques in Android provide options for balancing speed, effi ciency, and robustness.
    - **Shared Preferences** — When storing UI state, user preferences, or application settings, you want a lightweight mechanism to store a known set of values. Shared Preferences let you save groups of name/value pairs of primitive data as named preferences.
    - **Saved application UI state** — Activities and Fragments include specialized event handlers to record the current UI state when your application is moved to the background.
    - **Files** — It's not pretty, but sometimes writing to and reading from files is the only way to go. Android lets you create and load files on the device's internal or external media, providing support for temporary caches and storing fi les in publicly accessible folders.
  - Techniques support primitive types Boolean, string, float, long, and integer, making them ideal means of quickly storing default values, class instance variables, the current UI state, and user preferences.

- # CREATING AND SAVING SHARED PREFERENCES
  - Using the SharedPreferences class, you can create named maps of name/value pairs that can be persisted across sessions and shared among application components running within the same application sandbox.
  - To create or modify a Shared Preference, call getSharedPreferences on the current Context, passing in the name of the Shared Preference to change.
  - `SharedPreferences mySharedPreferences = getSharedPreferences(MY_PREFS, Activity.MODE_PRIVATE);`
  - Shared Preferences are stored within the application's sandbox, so they can be shared between an application's components but aren't available to other applications.
  - To modify a Shared Preference, use the SharedPreferences.Editor class. Get the Editor object by calling edit on the Shared Preferences object you want to change.
  - `SharedPreferences.Editor editor = mySharedPreferences.edit();`

- Use the put methods to insert or update the values associated with the specifi ed name:

```
// Store new primitive types in the shared preferences object.
editor.putBoolean(“isTrue”, true);
editor.putFloat(“lastFloat”, 1f);
editor.putInt(“wholeNumber”, 2);
editor.putLong(“aNumber”, 3l);
editor.putString(“textEntryValue”, “Not Empty”);
```

- To save edits, call apply or commit on the Editor object to save the changes asynchronously or synchronously, respectively.

```
// Commit the changes.
editor.apply();
```

- # RETRIEVING SHARED PREFERENCES
  - Accessing Shared Preferences, like editing and saving them, is done using the getSharedPreferences method.
  - Use the type-safe get methods to extract saved values.

```
// Retrieve the saved values.
boolean isTrue = mySharedPreferences.getBoolean(“isTrue”, false);
float lastFloat = mySharedPreferences.getFloat(“lastFloat”, 0f);
int wholeNumber = mySharedPreferences.getInt(“wholeNumber”, 1);
long aNumber = mySharedPreferences.getLong(“aNumber”, 0);
String stringPreference =
 mySharedPreferences.getString(“textEntryValue”, “”);
```

  - You can return a map of all the available Shared Preferences keys values by calling getAll, and check for the existence of a particular key by calling the contains method.

```
Map allPreferences = mySharedPreferences.getAll();
boolean containsLastFloat = mySharedPreferences.contains(“lastFloat”);
```

- INTRODUCING THE PREFERENCE FRAMEWORK AND THE PREFERENCE ACTIVITY
    - Android offers an XML-driven framework to create system-style Preference Screens for your applications.
    - By using this framework you can create Preference Activities that are consistent with those used in both native and other third-party applications
    - This has two distinct advantages:
        - Users will be familiar with the layout and use of your settings screens
        - You can integrate settings screens from other applications (including system settings such as location settings) into your application’s preferences.

- The preference framework consists of four parts:
  - **Preference Screen layout** — An XML file that defines the hierarchy of items displayed in your Preference screens. It specifies the text and associated controls to display, the allowed values, and the Shared Preference keys to use for each control.
  - **Preference Activity and Preference Fragment** — Extensions of Preference Activity and Preference Fragment respectively, that are used to host the Preference Screens. Prior to Android 3.0, Preference Activities hosted the Preference Screen directly; since then, Preference Screens are hosted by Preference Fragments, which, in turn, are hosted by Preference Activities.
  - **Preference Header definition** — An XML file that defines the Preference Fragments for your application and the hierarchy that should be used to display them.
  - **Shared Preference Change Listener** — An implementation of the **OnSharedPreferenceChangeListener** class used to listen for changes to Shared Preferences.

- Defining a Preference Screen Layout in XML
  - “Building User Interfaces,” Preference Screen layouts use a specialized set of controls designed specifi cally for preferences.
  - Each preference layout is defi ned as a hierarchy, beginning with a single PreferenceScreen element:

```
<?xml version=”1.0” encoding=”utf-8”?>
<PreferenceScreen
 xmlns:android=”http://schemas.android.com/apk/res/android”>
</PreferenceScreen>
```

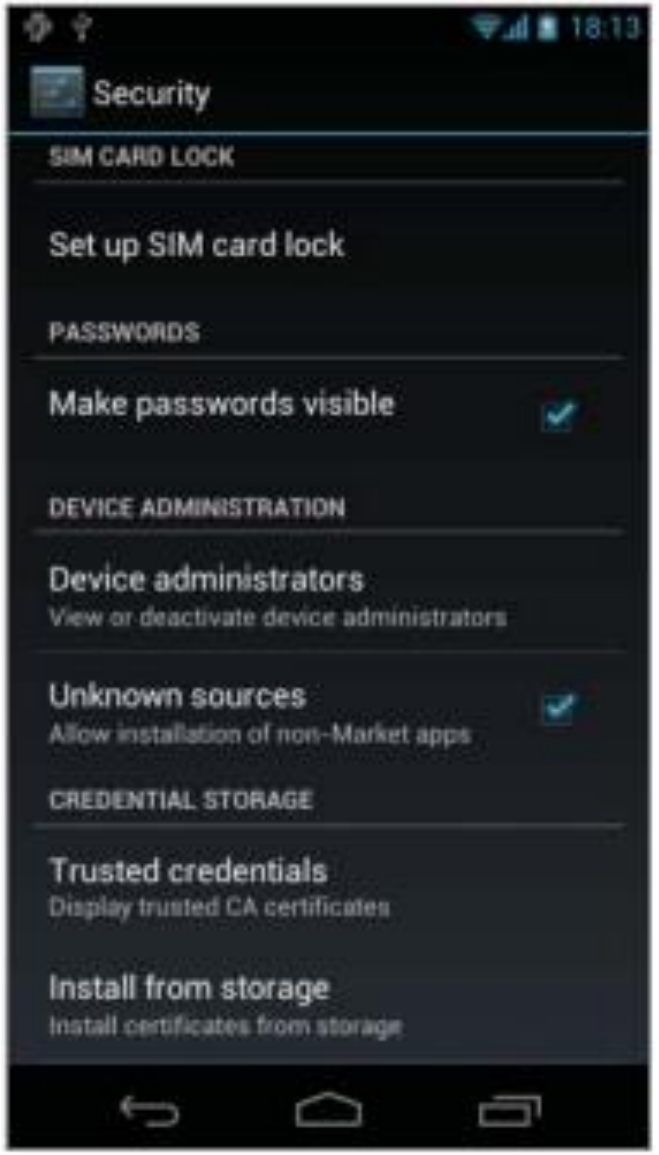
  - Within each Preference Screen you can include any combination of PreferenceCategory and Preference elements.
  - Preference Category elements, as shown in the following snippet, are used to break each Preference Screen into subcategories using a title bar separator:

```
<PreferenceCategory
 android:title=”My Preference Category”/>
```

- SIM card lock, device administration, and credential storage Preference Categories used on the Security Preference Screen.

- Although the specific attributes available for each preference control vary, each of them includes at least the following four:

- ‰ android:key — The Shared Preference key against which the selected value will be recorded.
- ‰ android:title — The text displayed to represent the preference.
- ‰ android:summary — The longer text description displayed in a smaller font below the title text.
- ‰ android:defaultValue — The default value that will be displayed (and selected) if no preference value has been assigned to the associated preference key



LISTING 7-1: A simple Shared Preferences screen

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory
    android:title="My Preference Category">
    <CheckBoxPreference
      android:key="PREF_CHECK_BOX"
      android:title="Check Box Preference"
      android:summary="Check Box Preference Description"
      android:defaultValue="true"
    />
  </PreferenceCategory>
</PreferenceScreen>
```

- Native Preference Controls
- Android includes several preference controls to build your Preference Screens:

  - `CheckBoxPreference` — A standard preference check box control used to set preferences to true or false.
  - `EditTextPreference` — Allows users to enter a string value as a preference. Selecting the preference text at run time will display a text-entry dialog.
  - `ListPreference` — The preference equivalent of a spinner. Selecting this preference will display a dialog box containing a list of values from which to select. You can specify different arrays to contain the display text and selection values.
  - `MultiSelectListPreference` — Introduced in Android 3.0 (API level 11), this is the preference equivalent of a check box list.
  - `RingtonePreference` — A specialized List Preference that presents the list of available ringtones for user selection. This is particularly useful when you're constructing a screen to configure notification settings.

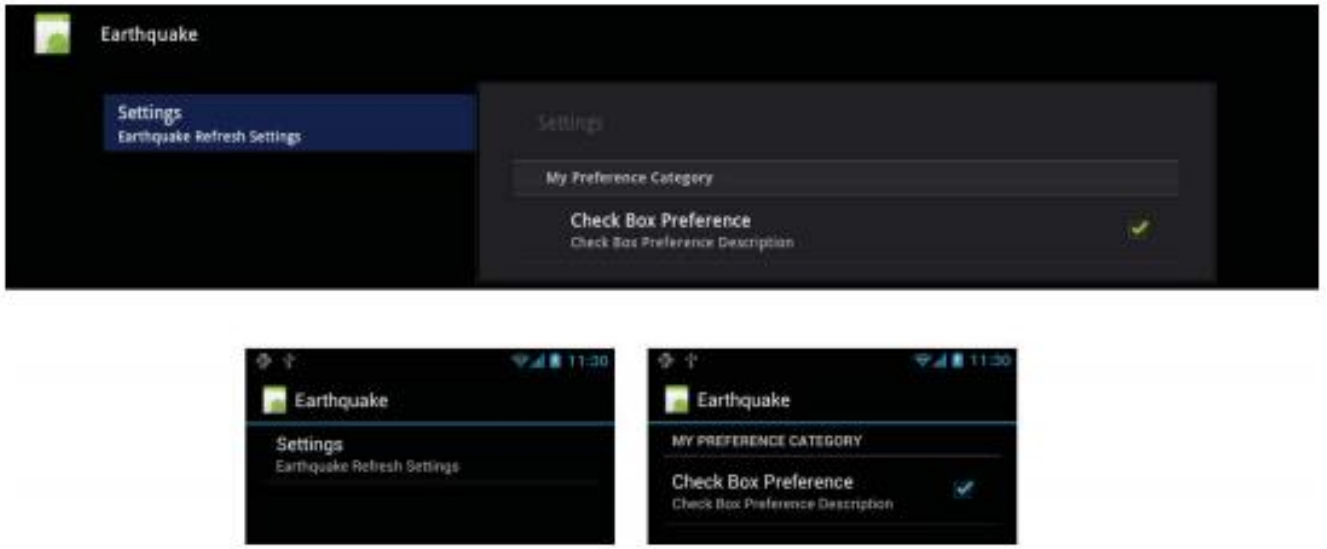
- Introducing the Preference Fragment
  - Since Android 3.0, the PreferenceFragment class has been used to host the preference screens defi ned by Preferences Screen resources. To create a new Preference Fragment, extend the PreferenceFragment class, as follows:

    - public class MyPreferenceFragment extends PreferenceFragment

  - To infl ate the preferences, override the onCreate handler and call addPreferencesFromResource, as shown here:

```
@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 addPreferencesFromResource(R.xml.userpreferences);
}
```

- Defining the Preference Fragment Hierarchy Using Preference Headers
  - Preference headers are XML resources that describe how your Preference Fragments should be grouped and displayed within a Preference Activity.
  - Each header identifies and allows you to select a particular Preference Fragment.



  - Preference Headers are XML resources stored in the res/xml folder of your project hierarchy. The resource ID for each header is the fi lename (without extension).

- Each header must be associated with a particular Preference Fragment that will be displayed when its header is selected. You must also specify a title and, optionally, a summary and icon resource to represent each Fragment and the Preference Screen it contains

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
  <header android:fragment="com.paad.preferences.MyPreferenceFragment"
          android:icon="@drawable/preference_icon"
          android:title="My Preferences"
          android:summary="Description of these preferences" />
</preference-headers>
```

- Like Preference Screens, you can invoke any Activity within your Preference Headers using an Intent. If you add an Intent node within a header element, as shown in the following snippet, the system will interpret this as a request to call startActivity using the specifi ed action

```
<header android:icon="@drawable/ic_settings_display"
        android:title="Intent"
        android:summary="Launches an Intent.">
 <intent android:action="android.settings.DISPLAY_SETTINGS "/>
</header>
```

- Introducing the Preference Activity
  - The PreferenceActivity class is used to host the Preference Fragment hierarchy defined by a preference headers resource. Prior to Android 3.0, the Preference Activity was used to host Preference Screens directly. For applications that target devices prior to Android 3.0, you may still need to use the Preference Activity in this way.

  - To create a new Preference Activity, extend the PreferenceActivity class as follows:
    - public class MyFragmentPreferenceActivity extends PreferenceActivity

  - When using Preference Fragments and headers, override the **onBuildHeaders** handler, calling **loadHeadersFromResource** and specifying your preference headers resource file:

```
public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.userpreferenceheaders, target);
}
```

- For legacy applications, you can inflate the Preference Screen directly in the same way as you would from a Preference Fragment — by overriding the onCreate handler and calling **addPreferencesFromResource**, specifying the Preference Screen layout XML resource to display within that Activity:

```
@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 addPreferencesFromResource(R.xml.userpreferences);
}
```

- Like all Activities, the Preference Activity must be included in the application manifest:

```
<activity android:name=".MyPreferenceActivity"
 android:label="My Preferences">
</activity>
```

- To display the application settings hosted in this Activity, open it by calling **startActivity** or **startActivityForResult**:

```
Intent i = new Intent(this, MyPreferenceActivity.class);
startActivityForResult(i, SHOW_PREFERENCES);
```

- # INCLUDING STATIC FILES AS RESOURCES
  - If your application requires external file resources, you can include them in your distribution package by placing them in the res/raw folder of your project hierarchy.
  - To access these read-only file resources, call the openRawResource method from your application's Resource object to receive an InputStream based on the specified file.
  - Pass in the filename (without the extension) as the variable name from the R.raw class

```
Resources myResources = getResources();
InputStream myFile = myResources.openRawResource(R.raw.myfilename);
```

  - Adding raw fi les to your resources hierarchy is an excellent alternative for large, preexisting data sources (such as dictionaries) for which it's not desirable (or even possible) to convert them into Android databases.

- WORKING WITH THE FILE SYSTEM
  - It’s good practice to use Shared Preferences or a database to store your application data, but there may still be times when you’ll want to use files directly rather than rely on Android’s managed mechanisms — particularly when working with multimedia files

  - File-Management Tools
    - Android supplies some basic fi le-management tools to help you deal with the fi le system. Many of these utilities are located within the java.io.File package.
    - Android does supply some specialized utilities for fi le management that are available from the application Context.

      - deleteFile — Enables you to remove fi les created by the current application
      - fileList — Returns a string array that includes all the fi les created by the current
      - Application

    - These methods are particularly useful for cleaning up temporary fi les left behind if your application crashes or is killed unexpectedly.

- Using Application-Specific Folders to Store Files
  - Many applications will create or download fi les that are specific to the application. There are two options for storing these application-specific fi les: internally or externally
  - Android offers two corresponding methods via the application Context, getDir and getExternalFilesDir, both of which return a File object that contains the path to the internal and external application fi le storage directory, respectively.
  - All fi les stored in these directories or the subfolders will be erased when your application is uninstalled.
  - In Android 2.2 (API level 8) the Environment class introduced a number of DIRECTORY_[Category] string constants that represent standard directory names, including downloads, images, movies, music, and camera files.

- Creating Private Application Files
  - Android offers the openFileInput and openFileOutput methods to simplify reading and writing streams from and to fi les stored in the application's sandbox.

```
String FILE_NAME = "tempfile.tmp";
// Create a new output file stream that's private to this application.
FileOutputStream fos = openFileOutput(FILE_NAME, Context.MODE_PRIVATE);
// Create a new file input stream.
FileInputStream fis = openFileInput(FILE_NAME);
```

  - These methods support only those fi les in the current application folder; specifying path separators will cause an exception to be thrown.

  - If the filename you specify when creating a FileOutputStream does not exist, Android will create it for you. The default behavior for existing files is to overwrite them; to append an existing file, specify the mode as Context.MODE_APPEND.

- The standard way to share a file between applications is to use a Content Provider. Alternatively, you can specify either Context.MODE_WORLD_READABLE or Context.MODE_WORLD_WRITEABLE when creating the output fi le, to make it available in other applications.

```
String OUTPUT_FILE = “publicCopy.txt”;
FileOutputStream fos = openFileOutput(OUTPUT_FILE, Context.MODE_WORLD_WRITEABLE);
```

- You can fi nd the location of fi les stored in your sandbox by calling getFilesDir. This method will return the absolute path to the fi les created using openFileOutput:

```
File file = getFilesDir();
Log.d(“OUTPUT_PATH_”, file.getAbsolutePath());
```

- Using the Application File Cache
  - Should your application need to cache temporary fi les, Android offers both a managed internal cache, and (since Android API level 8) an unmanaged external cache. You can access them by calling the getCacheDir and getExternalCacheDir methods, respectively, from the current Context.

  - Files stored in either cache location will be erased when the application is uninstalled.

- Storing Publicly Readable Files
  - Android 2.2 (API level 8) also includes a convenience method, Environment.getExternal StoragePublicDirectory, that can be used to find a path in which to store your application files.
  - The returned location is where users will typically place and manage their own fi les of each type
  - This is particularly useful for applications that provide functionality that replaces or augments system applications, such as the camera, that store fi les in standard locations.
  - The getExternalStoragePublicDirectory method accepts a String parameter that determines which subdirectory you want to access using a series of Environment static constants:
    - ‰ DIRECTORY_ALARMS — Audio files that should be available as user-selectable alarm sounds
    - ‰ DIRECTORY_DCIM — Pictures and videos taken by the device
    - ‰ DIRECTORY_DOWNLOADS — Files downloaded by the user
    - ‰ DIRECTORY_MOVIES — Movies
    - ‰ DIRECTORY_MUSIC — Audio fi les that represent music
    - ‰ DIRECTORY_NOTIFICATIONS — Audio files that should be available as user-selectable notification sounds
    - ‰ DIRECTORY_PICTURES — Pictures
    - ‰ DIRECTORY_PODCASTS — Audio fi les that represent podcasts
    - ‰ DIRECTORY_RINGTONES — Audio fi les that should be available as user-selectable ringtones

- Note that if the returned directory doesn’t exit, you must create it before writing fi les to the directory, as shown in the following snippet:

```
String FILE_NAME = “MyMusic.mp3”;
File path = Environment.getExternalStoragePublicDirectory(
                        Environment.DIRECTORY_MUSIC);
File file = new File(path, FILE_NAME);
try {
        path.mkdirs();
        [... Write Files ...]
} catch (IOException e) {
        Log.d(TAG, “Error writing “ + FILE_NAME, e);
}
```

- # INTRODUCING ANDROID DATABASES
  - Android provides structured data persistence through a combination of SQLite databases and Content Providers.
  - SQLite databases can be used to store application data using a managed, structured approach. Android offers a full SQLite relational database library.
  - Every application can create its own databases over which it has complete control.
  - Content Providers offer a generic, well-defi ned interface for using and sharing data that provides a consistent abstraction from the underlying data source.
  - Android databases are stored in the /data/data//databases folder on your device (or emulator). All databases are private, accessible only by the application that created them.
  - Content Providers provide an interface for publishing and consuming data, based around a simple URI addressing model using the content:// schema.
  - Content Providers can be shared between applications, queried for results, have their existing records updated or deleted, and have new records added.

- INTRODUCING SQLITE
  - SQLite is a well-regarded relational database management system (RDBMS). It is:
    - ‰ Open-source
    - ‰ Standards-compliant
    - ‰ Lightweight
    - ‰ Single-tier
  - It has been implemented as a compact C library that’s included as part of the Android software stack.
  - Each SQLite database is an integrated part of the application that created it. This reduces external dependencies, minimizes latency, and simplifi es transaction locking and synchronization.

- CONTENT VALUES AND CURSORS
  - Content Values are used to insert new rows into tables. Each ContentValues object represents a single table row as a map of column names to values.

  - Database queries are returned as Cursor objects. Cursors are pointers to the result set within the underlying data. Cursors provide a managed way of controlling your position (row) in the result set of a database query.

  - The *Cursor* class includes a number of navigation functions
    - *moveToFirst* — Moves the cursor to the first row in the query result
    - *moveToNext* — Moves the cursor to the next row
    - *moveToPrevious* — Moves the cursor to the previous row
    - *getCount* — Returns the number of rows in the result set
    - *getColumnIndexOrThrow* — Returns the zero-based index for the column with the specified name (throwing an exception if no column exists with that name)
    - *getColumnName* — Returns the name of the specified column index
    - *getColumnNames* — Returns a string array of all the column names in the current Cursor
    - *moveToPosition* — Moves the cursor to the specified row
    - *getPosition* — Returns the current cursor position

- # WORKING WITH SQLITE DATABASES
  - When working with databases, it's good form to encapsulate the underlying database and expose only the public methods and constants required to interact with that database, generally using what's often referred to as a contract or helper class.
  - This class should expose database constants, particularly column names, which will be required for populating and querying the database.
  - type of database constants that should be made public within a helper class.

```
// The index (key) column name for use in where clauses.
public static final String KEY_ID = "_id";
// The name and column index of each column in your database.
// These should be descriptive.
public static final String KEY_GOLD_HOARD_NAME_COLUMN = "GOLD_HOARD_NAME_COLUMN";
public static final String KEY_GOLD_HOARD_ACCESSIBLE_COLUMN = "OLD_HOARD_ACCESSIBLE_COLUMN";
public static final String KEY_GOLD_HOARDED_COLUMN = "GOLD_HOARDED_COLUMN";
// TODO: Create public field for each column in your table.
```

- Introducing the SQLiteOpenHelper
  - SQLiteOpenHelper is an abstract class used to implement the best practice pattern for creating, opening, and upgrading databases.
  - The SQLite Open Helper caches database instances after they've been successfully opened, so you can make requests to open the database immediately prior to performing a query or transaction.
  - Below code shows how to extend the SQLiteOpenHelper class by overriding the constructor, onCreate, and onUpgrade methods to handle the creation of a new database and upgrading to a new version, respectively.

```
private static class HoardDBOpenHelper extends SQLiteOpenHelper {

  private static final String DATABASE_NAME = "myDatabase.db";
  private static final String DATABASE_TABLE = "GoldHoards";
  private static final int DATABASE_VERSION = 1;

  // SQL Statement to create a new database.
  private static final String DATABASE_CREATE = "create table " +
    DATABASE_TABLE + " (" + KEY_ID +
    " integer primary key autoincrement, " +
    KEY_GOLD_HOARD_NAME_COLUMN + " text not null, " +
    KEY_GOLD_HOARDED_COLUMN + " float, " +
    KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + " integer);";

  public HoardDBOpenHelper(Context context, String name,
                    CursorFactory factory, int version) {
    super(context, name, factory, version);
  }

  // Called when no database exists in disk and the helper class needs
  // to create a new one.
  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(DATABASE_CREATE);
  }
```

```
  // Called when there is a database version mismatch meaning that
  // the version of the database on disk needs to be upgraded to
  // the current version.
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    // Log the version upgrade.
    Log.w("TaskDBAdapter", "Upgrading from version " +
      oldVersion + " to " +
      newVersion + ", which will destroy all old data");

    // Upgrade the existing database to conform to the new
    // version. Multiple previous versions can be handled by
    // comparing oldVersion and newVersion values.

    // The simplest case is to drop the old table and create a new one.
    db.execSQL("DROP TABLE IF IT EXISTS " + DATABASE_TABLE);
    // Create a new one.
    onCreate(db);
  }
}
```

- To access a database using the SQLite Open Helper, call getWritableDatabase or getReadableDatabase to open and obtain a writable or read-only instance of the underlying database, respectively.

- Behind the scenes, if the database doesn't exist, the helper executes its onCreate handler. If the database version has changed, the onUpgrade handler will fire.

- A call to getWritableDatabase can fail due to disk space or permission issues, so it's good practice to fall back to the getReadableDatabase method for database queries if necessary.

- In most cases this method will provide the same, cached writeable database instance as getWritableDatabase unless it does not yet exist or the same permission or disk space issues occur, in which case a read-only copy will be returned.

- Opening and Creating Databases Without the SQLite Open Helper
  - If you would prefer to manage the creation, opening, and version control of your databases directly, rather than using the SQLite Open Helper, you can use the application Context's *openOrCreateDatabase* method to create the database itself:

```
SQLiteDatabase db = context.openOrCreateDatabase(DATABASE_NAME, Context.MODE_PRIVATE, null);
```

  - After you have created the database, you must handle the creation and upgrade logic handled within the onCreate and onUpgrade handlers of the SQLite Open Helper — typically using the database's execSQL method to create and drop tables, as required.

- Querying a Database
  - Each database query is returned as a Cursor. This lets Android manage resources more effi ciently by retrieving and releasing row and column values on demand.
  - To execute a query on a Database object, use the query method, passing in the following:

    - An optional Boolean that specifies if the result set should contain only unique values.
    - The name of the table to query.
    - A projection, as an array of strings, that lists the columns to include in the result set.
    - A where clause that defines the rows to be returned. You can include ? wildcards that will be replaced by the values passed in through the selection argument parameter.
    - An array of selection argument strings that will replace the ? wildcards in the where clause.
    - A group by clause that defines how the resulting rows will be grouped.
    - A having clause that defines which row groups to include if you specified a group by clause.
    - A string that describes the order of the returned rows.
    - A string that defines the maximum number of rows in the result set.

- selection of rows from within an SQLite database table.

```
// Specify the result column projection. Return the minimum set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
  KEY_ID, KEY_GOLD_HOARD_ACCESSIBLE_COLUMN, KEY_GOLD_HOARDED_COLUMN };

// Specify the where clause that will limit our results.
String where = KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + "=" + 1;

// Replace these with valid SQL statements as necessary.
String whereArgs[] = null;
String groupBy = null;
String having = null;
String order = null;

SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
Cursor cursor = db.query(HoardDBOpenHelper.DATABASE_TABLE,
                         result_columns, where,
                         whereArgs, groupBy, having, order);
```

- Extracting Values from a Cursor
  - To extract values from a Cursor, first use the moveTo<location> methods described earlier to position the cursor at the correct row of the result Cursor, and then use the type-safe get<type> methods to return the value stored at the current row for the specified column.

  - To find the column index of a particular column within a result Cursor, use its getColumnIndexOrThrow and getColumnIndex methods.

```
int columnIndex = cursor.getColumnIndex(KEY_COLUMN_1_NAME);
if (columnIndex > -1) {
  String columnValue = cursor.getString(columnIndex);
  // Do something with the column value.
}
else {
  // Do something else if the column doesn't exist.
}
```

- The following code shows how to iterate over a result Cursor

```
float totalHoard = 0f;
float averageHoard = 0f;

// Find the index to the column(s) being used.
int GOLD_HOARDED_COLUMN_INDEX =
  cursor.getColumnIndexOrThrow(KEY_GOLD_HOARDED_COLUMN);

// Iterate over the cursors rows.
// The Cursor is initialized at before first, so we can
// check only if there is a "next" row available. If the
// result Cursor is empty this will return false.
while (cursor.moveToNext()) {
  float hoard = cursor.getFloat(GOLD_HOARDED_COLUMN_INDEX);
  totalHoard += hoard;
}

// Calculate an average -- checking for divide by zero errors.
float cursorCount = cursor.getCount();

averageHoard = cursorCount > 0 ?
                 (totalHoard / cursorCount) : Float.NaN;

// Close the Cursor when you've finished with it.
cursor.close();
```

- When you have finished using your result Cursor, it's important to close it to avoid memory leaks and reduce your application's resource load:
    - cursor.close();

- Adding, Updating, and Removing Rows
  - The SQLiteDatabase class exposes insert, delete, and update methods that encapsulate the SQL statements required to perform these actions.
  - Additionally, the execSQL method lets you execute any valid SQL statement on your database tables
  - Any time you modify the underlying database values, you should update your Cursors by running a new query.

- Inserting Rows
  - To create a new row, construct a ContentValues object and use its put methods to add name/value pairs representing each column name and its associated value.

  - Insert the new row by passing the Content Values into the insert method called on the target database — along with the table name

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Assign values for each row.
newValues.put(KEY_GOLD_HOARD_NAME_COLUMN, hoardName);
newValues.put(KEY_GOLD_HOARDED_COLUMN, hoardValue);
newValues.put(KEY_GOLD_HOARD_ACCESSIBLE_COLUMN, hoardAccessible);
// [ ... Repeat for each column / value pair ... ]

// Insert the row into your table
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.insert(HoardDBOpenHelper.DATABASE_TABLE, null, newValues);
```

- Updating Rows
  - Updating rows is also done with Content Values. Create a new ContentValues object, using the put methods to assign new values to each column you want to update. Call the update method on the database, passing in the table name, the updated Content Values object, and a where clause that specifies the row(s) to update

```
// Create the updated row Content Values.
ContentValues updatedValues = new ContentValues();

// Assign values for each row.
updatedValues.put(KEY_GOLD_HOARDED_COLUMN, newHoardValue);
// [ ... Repeat for each column to update ... ]

// Specify a where clause the defines which rows should be
// updated. Specify where arguments as necessary.
String where = KEY_ID + "=" + hoardId;
String whereArgs[] = null;

// Update the row with the specified index with the new values.
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.update(HoardDBOpenHelper.DATABASE_TABLE, updatedValues,
          where, whereArgs);
```

- Deleting Rows
  - To delete a row, simply call the delete method on a database, specifying the table name and a where clause that returns the rows you want to delete

```
// Specify a where clause that determines which row(s) to delete.
// Specify where arguments as necessary.
String where = KEY_GOLD_HOARDED_COLUMN + "=" + 0;
String whereArgs[] = null;

// Delete the rows that match the where clause.
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.delete(HoardDBOpenHelper.DATABASE_TABLE, where, whereArgs);
```

- # CREATING CONTENT PROVIDERS
  - Content Providers provide an interface for publishing data that will be consumed using a Content Resolver.
  - They allow you to decouple the application components that consume data from their underlying data sources, providing a generic mechanism through which applications can share their data or consume data provided by others.
  - To create a new Content Provider, extend the abstract ContentProvider class:

    ```
    public class MyContentProvider extends ContentProvider
    ```

  - it's good practice to include static database constants — particularly column names and the Content Provider authority — that will be required for transacting with, and querying, the database.
  - You will also need to override the onCreate handler to initialize the underlying data source, as well as the query, update, delete, insert, and getType methods to implement the interface used by the Content Resolver to interact with the data.

- Registering Content Providers
  - Like Activities and Services, Content Providers must be registered in your application manifest before the Content Resolver can discover them. This is done using a provider tag that includes a name attribute describing the Provider's class name and an authorities tag.

  - Use the authorities tag to define the base URI of the Provider's authority. A Content Provider's authority is used by the Content Resolver as an address and used to find the database you want to interact with.

  - Each Content Provider authority must be unique, so it's good practice to base the URI path on your package name.

  - The general form for defining a Content Provider's authority is as follows:
    - com.<CompanyName>.provider.<ApplicationName>
  - The completed provider tag should follow the format show in the following XML snippet:
    - <provider android:name=".MyContentProvider"
    - android:authorities="com.paad.skeletondatabaseprovider"/>

- Publishing Your Content Provider’s URI Address
  - Each Content Provider should expose its authority using a public static CONTENT_URI property to make it more easily discoverable. This should include a data path to the primary content — for example:
    - public static final Uri CONTENT_URI = Uri.parse(“content://com.paad.skeletondatabaseprovider/elements”);
  - These content URIs will be used when accessing your Content Provider using a Content Resolver.