Introduction to Mobile Programming

Android Programming

Chapter 3

Data and File Storage Overview

- **App-specific storage:** Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within external storage. Use the directories within internal storage to save sensitive information that other apps shouldn't access.

- **Shared storage:** Store files that your app intends to share with other apps, including media, documents, and other files.

- **Preferences:** Store private, primitive data in key-value pairs.

- **Databases:** Store structured data in a private database using the Room persistence library.

Data and File Storage Overview

How much space does your data require?

Internal storage has limited space for app-specific data. Use other types of storage if you need to save a substantial amount of data.

How reliable does data access need to be?

If your app's basic functionality requires certain data, such as when your app is starting up, place the data within internal storage directory or a database. App-specific files that are stored in external storage aren't always accessible because some devices allow users to remove a physical device that corresponds to external storage.

What kind of data do you need to store?

If you have data that's only meaningful for your app, use app-specific storage. For shareable media content, use shared storage so that other apps can access the content. For structured data, use either preferences (for key-value data) or a database (for data that contains more than 2 columns).

**Should the data be private to your app?**

When storing sensitive data—data that shouldn't be accessible from any other app—use internal storage, preferences, or a database. Internal storage has the added benefit of the data being hidden from users.

Data and File Storage Overview

	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	Yes
Media	Shareable media files (images, audio files, videos)	`MediaStore` API	READ_EXTERNAL_STORAGE when accessing other apps' files on Android 11 (API level 30) or higher READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE when accessing other apps' files on Android 10 (API level 29) 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

View on-device Files with Device File Explorer

① Click **View > Tool Windows > Device File Explorer** or click the **Device File Explorer** ▢ button in the tool window bar to open the Device File Explorer.
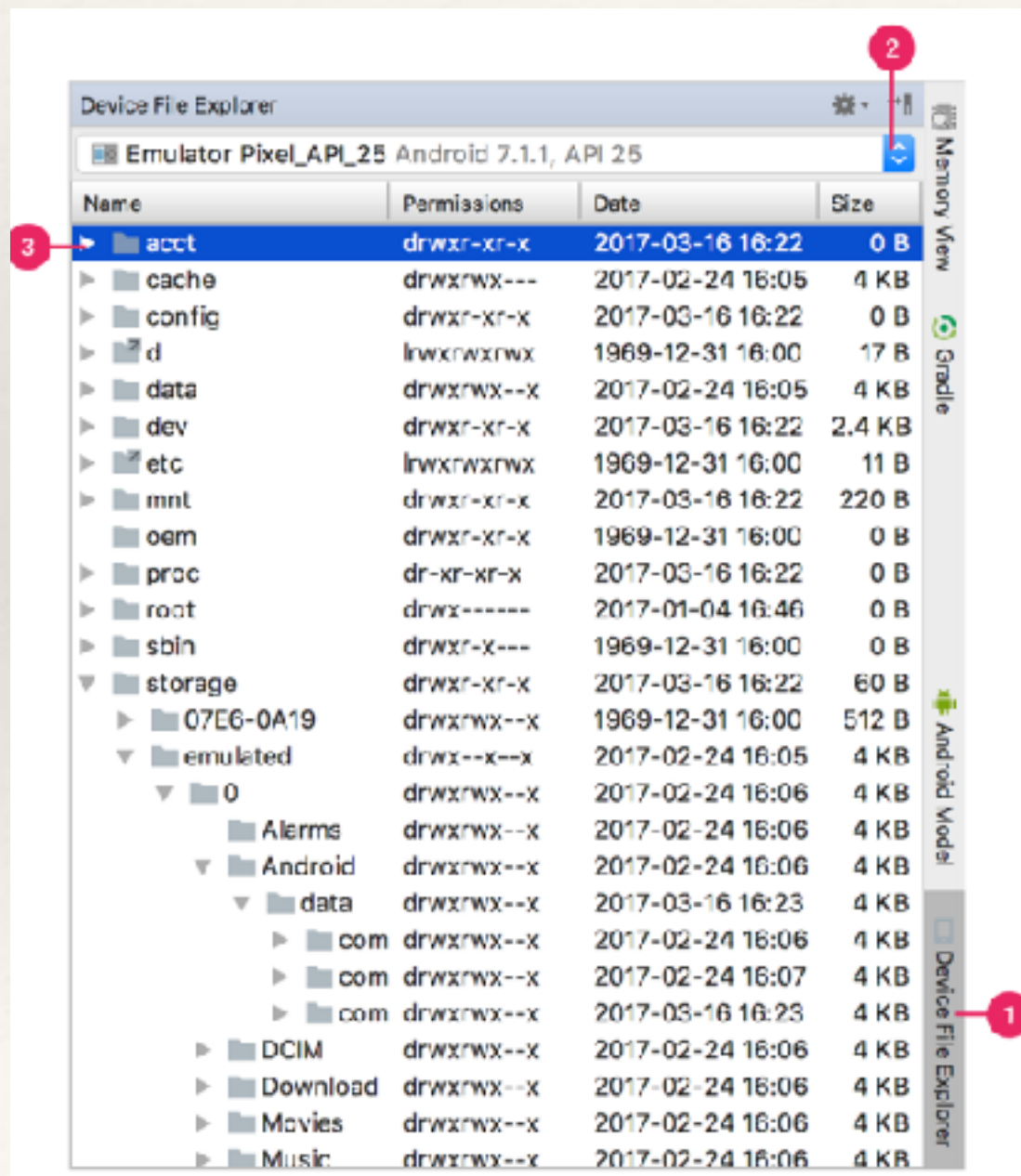
When exploring a device's files, the following directories are particularly useful:

data/data/*app_name*/

Contains data files for your app stored on internal storage

sdcard/

Contains user files stored on external user storage (pictures, etc.)

Data and File Storage Overview

- Internal file storage: Store app-private files on the device file system.
- External file storage: Store files on the shared external file system. This is usually for shared user files, such as photos.
- Shared preferences: Store private primitive data in key-value pairs.
- Databases: Store structured data in a private database.

1. If you want to expose your app's data to other apps, you can use a ContentProvider.

2. By default, files saved to the internal storage are private to your app, and other apps cannot access them.

3. When the user uninstalls your app, the files saved on the internal storage are removed.

4. If you'd like to keep some data temporarily, rather than store it persistently, you should use the special cache directory to save the data.

5. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

Best Practices for Operating on Files

1. Don't open and close files repeatedly

2. Share individual files

    1. FileProvider

    2. Content Provider

⚠ **Caution:** The exact location of where your files can be saved might vary across devices. For this reason, don't use hard-coded file paths.

3. Device File Explorer

```
<manifest ...
  android:installLocation="preferExternal">
  ...
</manifest>
```

⭐ **Note:** If your app requests a storage-related permission at runtime, the user-facing dialog indicates that your app is requesting broad access to external storage, even when scoped storage is enabled.

Access App-Specific Files

- **Internal storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. The system prevents other apps from accessing these locations, and on Android 10 (API level 29) and higher, these locations are encrypted. These characteristics make these locations a good place to store sensitive data that only your app itself can access.

- **External storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. Although it's possible for another app to access these directories if that app has the proper permissions, the files stored in these directories are meant for use only by your app. If you specifically intend to create files that other apps should be able to access, your app should store these files in the shared storage part of external storage instead.

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

```
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MODE_PRIVATE)) {
    fos.write(fileContents.toByteArray());
}
```

Data and File Storage Overview – II

- **Internal file storage**: Store app-private files on the device file system.
- **External file storage**: Store files on the shared external file system. This is usually for shared user files, such as photos.
- **Shared preferences**: Store private primitive data in key-value pairs.
- **Databases**: Store structured data in a private database.

1. If you don't need to store a lot of data and it doesn't require structure, you should use SharedPreferences.

2. The SharedPreferences APIs allow you to read and write persistent key-value pairs of primitive data types: booleans, floats, ints, longs, and strings.

3. The key-value pairs are written to XML files that persist across user sessions, even if your app is killed.

4. You can manually specify a name for the file or use per-activity files to save your data.

Databases

- Internal file storage: Store app-private files on the device file system.
- External file storage: Store files on the shared external file system. This is usually for shared user files, such as photos.
- Shared preferences: Store private primitive data in key-value pairs.
- Databases: Store structured data in a private database.

1. Android provides full support for SQLite databases.

2. Any database you create is accessible only by your app.

3. Instead of using SQLite APIs directly, it is recommended that you create and interact with your databases with the Room persistence library

4. The Room library provides an object-mapping abstraction layer that allows fluent database access while harnessing the full power of SQLite.

Internal Storage vs. External Storage

Internal storage:

- 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.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- 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.

1. Although apps are installed onto the internal storage by default, you can allow your app to be installed on external storage by specifying the android:installLocation attribute in your manifest.

2. Users appreciate this option when the APK size is very large and they have an external storage space that's larger than the internal storage.

Read and Write Operations

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



String filename = "myfile";
String fileContents = "Hello world!";
FileOutputStream outputStream;


try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write(fileContents.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Read and Write Operations

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

```java
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MODE_PRIVATE)) {
    fos.write(fileContents.toByteArray());
}
```

```java
FileInputStream fis = context.openFileInput(filename);
InputStreamReader inputStreamReader =
        new InputStreamReader(fis, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
try (BufferedReader reader = new BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
} catch (IOException e) {
    // Error occurred when opening raw file for reading.
} finally {
    String contents = stringBuilder.toString();
}
```

Cache File Operations

```java
File.createTempFile(filename, null, context.getCacheDir());
```

```java
File cacheFile = new File(context.getCacheDir(), filename);
```

```java
cacheFile.delete();
```

```java
context.deleteFile(cacheFileName);
```

```java
Array<String> files = context.fileList();
```

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

Directory Operations

When saving a file to internal storage, you can acquire the appropriate directory as a `File` by calling one of two methods:

`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.

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

`getFilesDir()`

> Returns a `File` representing the directory on the file system that's uniquely associated with your app.

`getDir(name, mode)`

> Creates a new directory (or opens an existing directory) within your app's unique file system directory. This new directory appears inside the directory provided by `getFilesDir()`.

`getCacheDir()`

> Returns a `File` representing the cache directory on the file system that's uniquely associated with your app. This directory is meant for temporary files, and it should be cleaned up regularly. The system may delete files there if it runs low on disk space, so make sure you check for the existence of your cache files before reading them.

To create a new file in one of these directories, you can use the `File()` constructor, passing the `File` object provided by one of the above methods that specifies your internal storage directory. For example:

Physical Storage Location

```java
File[] externalStorageVolumes =
        ContextCompat.getExternalFilesDirs(getApplicationContext(), null);
File primaryExternalStorage = externalStorageVolumes[0];
```

```java
File externalCacheFile = new File(context.getExternalCacheDir(), filename);
```

```java
externalCacheFile.delete();
```

Storage Availability

```
StorageStatsManager.getFreeBytes() / StorageStatsManager.getTotalBytes()
```

```java
// App needs 10 MB within internal storage.
private static final long NUM_BYTES_NEEDED_FOR_MY_APP = 1024 * 1024 * 10L;

StorageManager storageManager =
        getApplicationContext().getSystemService(StorageManager.class);
UUID appSpecificInternalDirUuid = storageManager.getUuidForPath(getFilesDir());
long availableBytes =
        storageManager.getAllocatableBytes(appSpecificInternalDirUuid);
if (availableBytes >= NUM_BYTES_NEEDED_FOR_MY_APP) {
    storageManager.allocateBytes(
            appSpecificInternalDirUuid, NUM_BYTES_NEEDED_FOR_MY_APP);
} else {
    Intent storageIntent = new Intent();
    storageIntent.setAction(ACTION_MANAGE_STORAGE);
    // Display prompt to user, requesting that they choose files to remove.
}
```

Saving 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 should be saved as public files.

- Private files: Files that rightfully belong to your app and will be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they don't provide value to the user outside of your app.

⚠ **Caution:** The external storage might become unavailable if the user removes the SD card or connects the device to a computer. And the files are still visible to the user and other apps that have the READ_EXTERNAL_STORAGE permission. So if your app's functionality depends on these files or you need to completely restrict access, you should instead write your files to the internal storage.

```xml
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18" />
    ...
</manifest>
```

1. Using the external storage is great for files that you want to share with other apps or allow the user to access with a computer.

Verifying that External Storage is Available

```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;
}
```

```java
myFile.delete();

myContext.deleteFile(fileName);
```

```java
public File getPublicAlbumStorageDir(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;
}
```

```java
public File getPrivateAlbumStorageDir(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;
}
```

Important Notes

★ **Note:** You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an **IOException** if one occurs. You may need to do this if you don't know exactly how much space you need. For example, if you change the file's encoding before you save it by converting a PNG image to JPEG, you won't know the file's size beforehand.

`getFreeSpace()` `getTotalSpace()`

★ **Note:** When the user uninstalls your app, the Android system deletes the following:

- All files you saved on internal storage.
- All files you saved external storage using **getExternalFilesDir()**.

However, you should manually delete all cached files created with **getCacheDir()** on a regular basis and also regularly delete other files you no longer need.

Overview of Shared Storage

- **Media content:** The system provides standard public directories for these kinds of files, so the user has a common location for all their photos, another common location for all their music and audio files, and so on. Your app can access this content using the platform's `MediaStore` API.

- **Documents and other files:** The system has a special directory for containing other file types, such as PDF documents and books that use the EPUB format. Your app can access these files using the platform's Storage Access Framework.

- **Datasets:** On Android 11 (API level 30) and higher, the system caches large datasets that multiple apps might use. These datasets can support use cases like machine learning and media playback. Apps can access these shared datasets using the `BlobStoreManager` API.

SharedPreferences

1. If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs.

2. A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them.

3. Each SharedPreferences file is managed by the framework and can be private or shared.

- 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.

SharedPreferences

SHARED PREFERENCES	SAVED INSTANCE STATE
Persist Data across user sessions, even if app is killed and restarted, or device is rebooted	Preserves state data across activity instances in same user session.
Data that should be remembered across sessions, such as user's preferred settings or their game score.	Data that should not be remembered across sessions, such as currently selected tab or current state of activity.
Common use is to store user preferences	Common use is to recreate state after the device has been rotated

Read and Write for SharedPreferences

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

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

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

apply()

```java
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.integer.saved_high_score_default_key);
int highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue);
```

Methods of Shared Preferences

Methods of Shared Preferences

1. **contains(String key)**: This method is used to check whether the preferences contains a preference.
2. **edit()**: This method is used to create a new Editor for these preferences, through which you can make modifications to the data in the preferences and atomically commit those changes back to the SharedPreferences object.
3. **getAll()**: This method is used to retrieve all values from the preferences.
4. **getBoolean(String key, boolean defValue)**: This method is used to retrieve a boolean value from the preferences.
5. **getFloat(String key, float defValue)**: This method is used to retrieve a float value from the preferences.
6. **getInt(String key, int defValue)**: This method is used to retrieve an int value from the preferences.
7. **getLong(String key, long defValue)**: This method is used to retrieve a long value from the preferences.
8. **getString(String key, String defValue)**: This method is used to retrieve a String value from the preferences.
9. **getStringSet(String key, Set defValues)**: This method is used to retrieve a set of String values from the preferences.
10. **registerOnSharedPreferencechangeListener(SharedPreferences.OnSharedPreferencechangeListener listener)**: This method is used to registers a callback to be invoked when a change happens to a preference.
11. **unregisterOnSharedPreferencechangeListener(SharedPreferences.OnSharedPreferencechangeListener listener)**: This method is used to unregisters a previous callback.

Sending Simple Data to Other Apps

Send text content

```java
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

```java
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(Intent.createChooser(sendIntent, getResources().getText(R.string.send_to)));
```

Send binary content

```java
Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND);
shareIntent.putExtra(Intent.EXTRA_STREAM, uriToImage);
shareIntent.setType("image/jpeg");
startActivity(Intent.createChooser(shareIntent, getResources().getText(R.string.send_to)));
```

Sending Simple Data to Other Apps

Send multiple pieces of content

```java
ArrayList<Uri> imageUris = new ArrayList<Uri>();
imageUris.add(imageUri1); // Add your image URIs here
imageUris.add(imageUri2);

Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND_MULTIPLE);
shareIntent.putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageUris);
shareIntent.setType("image/*");
startActivity(Intent.createChooser(shareIntent, "Share images to.."));
```

Receiving Simple Data from Other Apps

```xml
<activity android:name=".ui.MyActivity" >
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND_MULTIPLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
</activity>
```

Caution: Take extra care to check the incoming data, you never know what some other application may send you. For example, the wrong MIME type might be set, or the image being sent might be extremely large. Also, remember to process binary data in a separate thread rather than the main ("UI") thread.

```java
void onCreate (Bundle savedInstanceState) {
    ...
    // Get intent, action and MIME type
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null) {
        if ("text/plain".equals(type)) {
            handleSendText(intent); // Handle text being sent
        } else if (type.startsWith("image/")) {
            handleSendImage(intent); // Handle single image being sent
        }
    } else if (Intent.ACTION_SEND_MULTIPLE.equals(action) && type != null) {
        if (type.startsWith("image/")) {
            handleSendMultipleImages(intent); // Handle multiple images being sent
        }
    } else {
        // Handle other intents, such as being started from the home screen
    }
    ...
}

void handleSendText(Intent intent) {
    String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
    if (sharedText != null) {
        // Update UI to reflect text being shared
    }
}

void handleSendImage(Intent intent) {
    Uri imageUri = (Uri) intent.getParcelableExtra(Intent.EXTRA_STREAM);
    if (imageUri != null) {
        // Update UI to reflect image being shared
    }
}

void handleSendMultipleImages(Intent intent) {
    ArrayList<Uri> imageUris = intent.getParcelableArrayListExtra(Intent.EXTRA_STREAM);
    if (imageUris != null) {
        // Update UI to reflect multiple images being shared
    }
}
```

Serializable vs. Parcelable

❖ Serializable is a standard Java interface.

  ❖ Simple

  ❖ More Garbage

  ❖ Slower

❖ Parcelable is a part of Android SDK.

  ❖ Complex

  ❖ Less Garbage

  ❖ Faster

  ❖ Custom Code

❖ Both of them are designed for sending objects over Intent object.

Implementing Parcelable

```java
public class Details {

    private String name;
    private String age;
    private String phone;

    public Details(String name, String age, String phone) {
        this.name = name;
        this.age = age;
        this.phone = phone;
     }
        public String getName() {
        return name;
    }

    public String getAge() {
        return age;
    }

    public String getPhone() {
        return phone;
    }

        }
```

Implementing Parcelable

```
1    package com.animeshroy.notekeeper.testapp;
2
3    import android.os.Parcel;
4    import android.os.Parcelable;
5
6    public class Details implements Parcelable {
7
8        private String name;
9        private String age;
10       private String phone;
11
12       public Details(String name, String age, String phone) {
13           this.name = name;
14           this.age = age;
15           this.phone = phone;
16       }
17
18       protected Details(Parcel in) {
19           name = in.readString();
20           age = in.readString();
21           phone = in.readString();
22       }
23
24       public static final Creator<Details> CREATOR = new Creator<Details>() {
25           @Override
26           public Details createFromParcel(Parcel in) {
27               return new Details(in);
28           }
29
30           @Override
31           public Details[] newArray(int size) {
32               return new Details[size];
33           }
34       };
35
36       public String getName() {
37           return name;
38       }
39
40       public String getAge() {
41           return age;
42       }
43
44       public String getPhone() {
45           return phone;
46       }
```

```
48       @Override
49       public int describeContents() {
50           return 0;
51       }
52
53       @Override
54       public void writeToParcel(Parcel dest, int flags) {
55           dest.writeString(name);
56           dest.writeString(age);
57           dest.writeString(phone);
58       }
59   }
```

How to pass Parcelable

```java
import android.content.Intent;
import android.os.Parcel;
import android.os.Parcelable;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class FirstActivity extends AppCompatActivity {

    EditText name;
    EditText age;
    EditText phone;
    Button button;
    Details details;

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

        name = findViewById(R.id.edit_name);
        age = findViewById(R.id.edit_age);
        phone = findViewById(R.id.edit_phone);
        button = findViewById(R.id.button);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                String name_str = name.getText().toString();
                String age_str = age.getText().toString();
                String phone_str = phone.getText().toString();

                if ((name_str.matches("")) ||(age_str.matches(""))|| (phone_str.matches(""))
                    Toast.makeText(FirstActivity.this, "You must fill all the field (Name, a
                        Toast.LENGTH_SHORT).show();
                    return;
                }
```

```java
                details = new Details(name_str, age_str, phone_str);

                Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
                intent.putExtra("Information", details);
                startActivity(intent);
            }
        });

    }
}
```

How to Get Parcelable

```java
import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;


public class SecondActivity extends AppCompatActivity {

    TextView text_name, text_age, text_phone;


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


        text_name = findViewById(R.id.name);
        text_age = findViewById(R.id.age);
        text_phone = findViewById(R.id.phone);



        Intent intent = getIntent();
        Details student = intent.getParcelableExtra("Information");


        String name = student.getName();
        String age = student.getAge();
        String phone = student.getPhone();


        text_name.setText("Your Name: "+name);
        text_age.setText("Your Age: "+age);
        text_phone.setText("Your Phone No: "+phone);


    }
}
```

Serialization in Android

```java
import java.io.Serializable;

public class Person implements Serializable //Added implements Serializable
{
    String name="";
    private String number="";
    private String address="";
    private static final long serialVersionUID = 46543445;

    public void setName(String name)
    {
        this.name = name;
    }

    public void setNumber(String number)
    {
        this.number = number;
    }

    public void setAddress(String address)
    {
        this.address = address;
    }

    public String getName()
    {
        return name;
    }

    public String getNumber()
    {
        return number;
    }

    public String getAddress()
    {
        return address;
    }
}
```

Saving Serialized Object

```java
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class SerializationDemo extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Person person = new Person();
        person.setName("CoderzHeaven");
        person.setAddress("CoderzHeaven India");
        person.setNumber("1234567890");

        //save the object
        saveObject(person);

        // Get the Object
        Person person1 = (Person)loadSerializedObject(new File("/sdcard/save_o
        System.out.println("Name : " + person1.getName());
    }

    public void saveObject(Person p){
        try
        {
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStre
            oos.writeObject(p); // write the class as an 'object'
            oos.flush(); // flush the stream to insure all of the information
            oos.close();// close the stream
        }
        catch(Exception ex)
        {
            Log.v("Serialization Save Error : ",ex.getMessage());
            ex.printStackTrace();
        }
    }

    public Object loadSerializedObject(File f)
    {
        try
        {
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
            Object o = ois.readObject();
            return o;
        }
        catch(Exception ex)
        {
        Log.v("Serialization Read Error : ",ex.getMessage());
            ex.printStackTrace();
        }
        return null;
    }
}
```