Exercise 5: External Storage

Due: Apr. 2, 23:59 PDT

Overview

This exercise provides an example for reading and writing files in Android external storage, shows the difference between app-specified external storage with shared external storage, and demonstrates how to request permissions at runtime.

Name the project as “Exercise5YourName”.

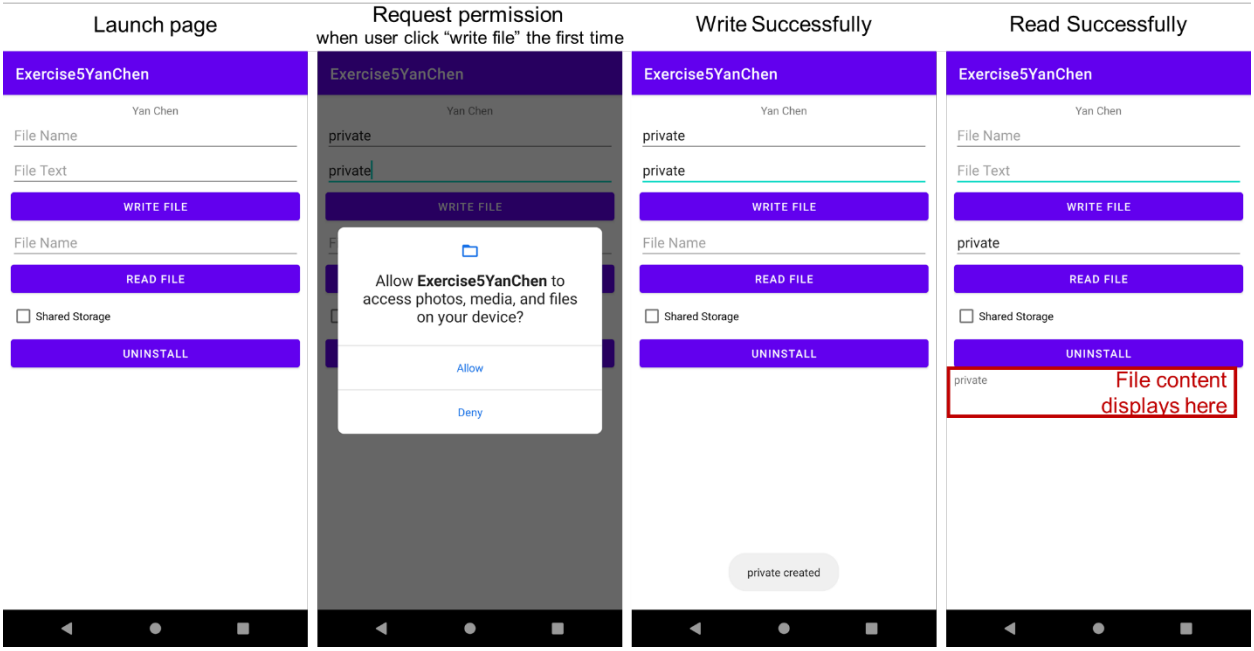

Fig. 1 Sample run of the app

Prerequisites:

- Know the process of submitting your work (exercise 0)
- Be familiar with previous topics related to project structure and UI (exercise 1 - 4, lesson 2 - 4)
- Has set up an emulator (API 29)

Procedure related to the above topics will not be provided in the instruction. Refer to corresponding exercise/lecture notes if needed.

If you set any different view id’s, filenames, etc., remember to modify the corresponding part of code.

Step 0. Register Manifest Permissions

We need to add permissions in manifest file to read and write files in external storage. However, from API 29, the default storage is “scoped storage”, which means the apps do NOT have direct access to all external files anymore (i.e., only has access to app-specified external storage).

“Fortunately,” we are focusing on API 29, and API 29 allows developers to opt-out scoped storage by setting requestLegacyExternalStorage attribute to be true. But note that, from API 30, scoped storage is mandatory for all apps. See the release note for storage and the blog of FAQ for more details.

Just to be safe, let’s set requestLegacyExternalStorage and the max API to API 29.

And add the permission to uninstall the app for testing if the files will still be there after uninstallation.

```
<manifest ...>
    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        android:maxSdkVersion="29"/>
    <uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission
        android:name="android.permission.REQUEST_DELETE_PACKAGES"/>
    <application
        ...
        android:requestLegacyExternalStorage="true">
        ...
    </application>
</manifest>
```

Step 1. Setup the UI

The positions of the widgets do not matter but you should include the elements listed below.

Maybe easier to use a vertical LinearLayout. And you can set android:importantForAutofill="no" and inputType="text" for all EditTexts to get rid of the warnings.

Also, when you set the onClick attribute for Buttons, the IDE will complain because we haven’t implemented the methods yet. After finishing the remaining steps, the error will be gone.

Widget	id	Text/Hint	onClick
TextView	n/a	android:text="Your name"	n/a
EditText	nameW	android:hint="File Name"	n/a
EditText	content	android:hint="File Text"	n/a
Button	n/a	android:text="Write File"	writeFile
EditText	nameR	android:hint="File Name"	n/a
Button	n/a	android:text="Read File"	readFile
CheckBox	share	android:text="Shared storage?"	n/a
Button	n/a	android:text="Uninstall"	uninstall
TextView	result	android:text=Will set later	n/a

Step 2. Implement a FileOperations class

Create a new Java class called FileOperations. It's a regular Java class, not an Activity class!

2.1 Check if External Storage is Available

Implement a private helper method to check if the external storage is available.

```
public static boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    return Environment.MEDIA_MOUNTED.equals(state);
}
```

2.2 Write to External Storage

Implement the method to write a file. The method takes three parameters, a file path, a String for file name, and a String for file content; returns true if file written successfully.

To write, first check if the external storage is available use the private helper method we implemented.

We also need to check if there is enough space. However, we don't know the file size, so we can simply wrap the code in a try/catch. You can use other Java.io classes to write the file.

Note that for simplicity, if the file already exists, the new content will replace the existing content. In a more realistic setting, you may want to display a dialog asking the user to choose if he/she wants to append to the existing file, or replace, or cancel the operation.

```
public static Boolean write(File path, String name, String content) {
    // Return false if external storage is not available
    if (!isExternalStorageWritable()) return false;

    // Don't know the file size, so wrap in try/catch
    try {
        File file = new File(path, name);
        // Can use other Java.io classes to write the file
        // New content will replace existing content if file exists
        FileWriter writer = new FileWriter(file);
        writer.write(content);
        writer.close();
        return true;
    } catch (IOException e) {
        return false;
    }
}
```

2.3 Read External Storage

Implement the method to read a file. The method takes the path and the name of the file as the parameters, and returns a String containing the content of the file being read.

Similar to write, you can use other classes in java.io package for reading a file.

Also, wrap the code in a try/catch for any potential IO error, such as file not exist.

```
public static String read(File path, String name) {
    try {
        StringBuilder output = new StringBuilder();
        File file = new File(path, name);
        // Can use other Java.io classes to read the file
        BufferedReader br = new BufferedReader(new FileReader(file));
        String line = "";
        while ((line = br.readLine()) != null) {
            output.append(line).append("\n");
        }
        return output.toString();
    } catch (IOException e) {
        return null;
    }
}
```

Step 3. Implement MainActivity

Other than the binding object or the View objects (depends on if you are using view binding or not), the MainActivity class has two attributes:

```
private static final String FILE_TYPE = Environment.DIRECTORY_DOWNLOADS;
private static final int REQUEST_CODE = 100;
```

3.1 Implement onCreate

Same as all activities, in onCreate, you need to initialize the views, or inflate a binding object. See exercise 1 & 2, lesson 3 page 11 - 12 if needed.

3.2 Get File Path

There are two types of external storage. One is app-specified (private), one is global (public). Let's implement a private helper method to get the paths of these two directories to test the behaviors of both types. Note that the method getExternalStoragePublicDirectory was deprecated since API 29 for privacy. A better practice now is to use a Content Provider, which we haven't covered it. Therefore, let's still use the deprecated method. Just ignore IDE's complaint.

```
private File getPath() {
    // If check box checked, store to shared storage
    if (binding.share.isChecked())
        return Environment.getExternalStoragePublicDirectory(FILE_TYPE);

    // If check box unchecked, store to app-specific storage
    else return this.getExternalFilesDir(FILE_TYPE);
}
```

3.3 Check Permission and Request Permission

As explained in class, writing/reading files from external storage is considered as “dangerous” permissions that you need to request from the user at runtime. Especially the writing.

So, implement a private helper method to check if Write permission is given:

```
private boolean checkPermission() {
    int result = ContextCompat.checkSelfPermission(MainActivity.this,
        android.Manifest.permission.WRITE_EXTERNAL_STORAGE);
    return result == PackageManager.PERMISSION_GRANTED;
}
```

Also, implement a private helper method to request the permission:

```
private void requestPermission() {
    ActivityCompat.requestPermissions(MainActivity.this,
        new String[]{android.Manifest.permission.WRITE_EXTERNAL_STORAGE},
        REQUEST_CODE);
}
```

Note that user may denied the permission, or even chose “Deny and never ask again”. To handle these situations, override the onRequestPermissionsResult method, which is called every time the requestPermissions method is executed. Here we just simply toast messages. In reality, you may display dialog explaining why the permission is needed, etc.

Note that the shouldShowRequestPermissionRationale is only available for API 23+ (M), and since we set the minimum SDK to 21, we need to add an annotation on the method (or change the minimum SDK to 23+, or wrap the code in an if statement).

```
@RequiresApi(api = Build.VERSION_CODES.M)
@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    // If user denied the permission
    if (requestCode == REQUEST_CODE
            && grantResults[0] == PackageManager.PERMISSION_DENIED) {
        // If user chose "Deny and never ask again"
        if (!shouldShowRequestPermissionRationale(permissions[0])) {
            Toast.makeText(this, "Permission denied",
                    Toast.LENGTH_SHORT).show();
        } else { // If user chose "Deny"
            Toast.makeText(this, "Please give writing permission",
                    Toast.LENGTH_SHORT).show();
        }
    }
}
```

3.4 Write Files

When user clicks the “Write File” button, a file is written based on the file name and the content user entered in the text boxes, to an external storage directory based on the user choice (save to shared storage or not). So, let’s implement the “writeFile” method we’ve set that is executed for an onClick event on “Write File” button.

The method first uses the helper method to check if the permission was given. If given, call the private helper method to get the path, and try to write the file based on user inputs. A message “... created” will be toasted if file successfully created.

Note that the try/catch in the write method of the FileOperations class would also catch those error due to invalid user input (such as empty input). Therefore, even without validating user input, the app will not crash on invalid user inputs. For demonstration purpose, the message “I/O error” will show for any errors. However, in a more realistic setting, it’s better to have various error messages accordingly.

```
public void writeFile(View v) {
    String name = binding.nameW.getText().toString();
    String content = binding.content.getText().toString();
    if (checkPermission()) {
        File path = getPath();
        // If successfully wrote
        if (FileOperations.write(path, name, content))
            Toast.makeText(this, name + " created",
                    Toast.LENGTH_SHORT).show();
        // If anything wrong
        else
            Toast.makeText(this, "I/O error",
                    Toast.LENGTH_SHORT).show();
    } else requestPermission();
}
```

3.5 Read Files

When user clicks the “Read File” button, the content of the file will be displayed based on the file name and user choice (read from shared storage or not). As set in step 1, the “Read File” button responds to an onClick event by “readFile” method.

Similarly, it may be better to differentiate various errors, but here, we only toast “File not found”.

```
public void readFile(View v) {
    String name = binding.nameR.getText().toString();
    File path = getPath();
    String text = FileOperations.read(path, name);
    // If read nothing, means something wrong
    if (text == null) {
        Toast.makeText(this, "File not Found",
                Toast.LENGTH_SHORT).show();
    }
    binding.result.setText(text);
}
```

3.6 Uninstall

For testing if the file created will be removed or not after uninstalling the app, we have a button to uninstall the app. Implement the “uninstall” method to respond to an onClick event.

Recall that we are using an implicit Intent for uninstalling the app.

```
public void uninstall(View v) {
    Intent delete = new Intent(Intent.ACTION_DELETE,
        Uri.parse("package:" + getPackageName()));
    startActivity(delete);
}
```

Step 4. Test the app

Write a file called “private” with the “Shared storage?” option unchecked. Then write a file called “public” with the “Shared storage?” option checked. Contents do not matter. Try to read the two files with the option unchecked/checked respectively.

Keep the app running and open “Device File Explorer” as shown in Lesson 15 page 8. The file named “private” should be under “sdcard/Android/data/edu.sjsu.android.exercise5yourname/files/Download” folder; the file named public should be under “sdcard/Download” folder.

Uninstall the app but don’t turn off the emulator. Refresh the Device File Explorer (right-click -> Synchronize). You should see the whole project folder under sdcard is removed, so does the “private” file. But the “public” file should still be there, under “sdcard/Download” folder.

Submission

- Push your project to a Bitbucket repository (name it “exercise5”) by the due date.
- Invite and share your Bitbucket repository the grader (edmond.lin@sjsu.edu) and the instructor (yan.chen01@sjsu.edu).
- Submit repository links, etc. by answering all the questions in the “Exercise 5 - External Storage” quiz on Canvas.
- Only your last submission before deadline will be graded based on the following criteria:
  2 pts if meets all requirements;
  1 pt if app failed/missing any requirement.