android developers

Notepad Tutorial:

Notepad Exercise 3

In this exercise, you will use life-cycle event callbacks to store and retrieve application state data. This exercise demonstrates:

- *Life-cycle events and how your application can use them*
- *Techniques for maintaining application state*

[Exercise 1] [Exercise 2] [Exercise 3] [Extra Credit]

Step 1

Import `Notepadv3` into Eclipse. If you see an error about `AndroidManifest.xml,` or some problems related to an Android zip file, right click on the project and select **Android Tools** > **Fix Project Properties** from the popup menu. The starting point for this exercise is exactly where we left off at the end of the Notepadv2.

The current application has some problems — hitting the back button when editing causes a crash, and anything else that happens during editing will cause the edits to be lost.

To fix this, we will move most of the functionality for creating and editing the note into the NoteEdit class, and introduce a full life cycle for editing notes.

1. Remove the code in `NoteEdit` that parses the title and body from the extras Bundle.

   Instead, we are going to use the `DBHelper` class to access the notes from the database directly. All we need passed into the NoteEdit Activity is a `mRowId` (but only if we are editing, if creating we pass nothing). Remove these lines:

   ```
   String title = extras.getString(NotesDbAdapter.KEY_TITLE);
   String body = extras.getString(NotesDbAdapter.KEY_BODY);
   ```

2. We will also get rid of the properties that were being passed in the `extras` Bundle, which we were using to set the title and body text edit values in the UI. So delete:

   ```
   if (title != null) {
       mTitleText.setText(title);
   }
   if (body != null) {
       mBodyText.setText(body);
   }
   ```

Step 2

Create a class field for a `NotesDbAdapter` at the top of the NoteEdit class:

```
    private NotesDbAdapter mDbHelper;
```

Also add an instance of `NotesDbAdapter` in the `onCreate()` method (right below the `super.onCreate()` call):

```
    mDbHelper = new NotesDbAdapter(this);

    mDbHelper.open();
```

Step 3

In `NoteEdit`, we need to check the *savedInstanceState* for the `mRowId`, in case the note editing contains a saved state in the Bundle, which we should recover (this would happen if our Activity lost focus and then restarted).

1. Replace the code that currently initializes the `mRowId`:

   ```
           mRowId = null;

           Bundle extras = getIntent().getExtras();
           if (extras != null) {
               mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);
           }
   ```

   with this:

   ```
           mRowId = (savedInstanceState == null) ? null :
               (Long) savedInstanceState.getSerializable(NotesDbAdapter.KEY_ROWID);
           if (mRowId == null) {
               Bundle extras = getIntent().getExtras();
               mRowId = extras != null ? extras.getLong(NotesDbAdapter.KEY_ROWID)
                                       : null;
           }
   ```

2. Note the null check for `savedInstanceState`, and we still need to load up `mRowId` from the `extras` Bundle if it is not provided by the `savedInstanceState`. This is a ternary operator shorthand to safely either use the value or null if it is not present.
3. Note the use of `Bundle.getSerializable()` instead of `Bundle.getLong()`. The latter encoding returns a `long` primitive and so can not be used to represent the case when `mRowId` is `null`.

Step 4

Next, we need to populate the fields based on the `mRowId` if we have it:

```
populateFields();
```

This goes before the `confirmButton.setOnClickListener()` line. We'll define this method in a moment.

Step 5

Get rid of the Bundle creation and Bundle value settings from the `onClick()` handler method. The Activity no longer needs to return any extra information to the caller. And because we no longer have an Intent to return, we'll use the shorter version of `setResult()`:

```
public void onClick(View view) {
    setResult(RESULT_OK);
    finish();
}
```

We will take care of storing the updates or new notes in the database ourselves, using the life-cycle methods.

The whole `onCreate()` method should now look like this:

```
super.onCreate(savedInstanceState);

mDbHelper = new NotesDbAdapter(this);
mDbHelper.open();

setContentView(R.layout.note_edit);

mTitleText = (EditText) findViewById(R.id.title);
mBodyText = (EditText) findViewById(R.id.body);

Button confirmButton = (Button) findViewById(R.id.confirm);

mRowId = (savedInstanceState == null) ? null :
    (Long) savedInstanceState.getSerializable(NotesDbAdapter.KEY_ROWID);
if (mRowId == null) {
    Bundle extras = getIntent().getExtras();
    mRowId = extras != null ? extras.getLong(NotesDbAdapter.KEY_ROWID)
                            : null;
}

populateFields();

confirmButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {
        setResult(RESULT_OK);
        finish();
    }

});
```

Step 6

Define the `populateFields()` method.

```
private void populateFields() {
    if (mRowId != null) {
        Cursor note = mDbHelper.fetchNote(mRowId);
        startManagingCursor(note);
        mTitleText.setText(note.getString(
                    note.getColumnIndexOrThrow(NotesDbAdapter.KEY_TITLE)));
```

```
            mBodyText.setText(note.getString(
                    note.getColumnIndexOrThrow(NotesDbAdapter.KEY_BODY)));
        }
    }
```

This method uses the `NotesDbAdapter.fetchNote()` method to find the right note to edit, then it calls `startManagingCursor()` from the `Activity` class, which is an Android convenience method provided to take care of the Cursor life-cycle. This will release and re-create resources as dictated by the Activity life-cycle, so we don't need to worry about doing that ourselves. After that, we just look up the title and body values from the Cursor and populate the View elements with them.

Step 7

Still in the `NoteEdit` class, we now override the methods `onSaveInstanceState()`, `onPause()` and `onResume()`. These are our life-cycle methods (along with `onCreate()` which we already have).

`onSaveInstanceState()` is called by Android if the Activity is being stopped and **may be killed before it is resumed!** This means it should store any state necessary to re-initialize to the same condition when the Activity is restarted. It is the counterpart to the `onCreate()` method, and in fact the `savedInstanceState` Bundle passed in to `onCreate()` is the same Bundle that you construct as `outState` in the `onSaveInstanceState()` method.

`onPause()` and `onResume()` are also complimentary methods. `onPause()` is always called when the Activity ends, even if we instigated that (with a `finish()` call for example). We will use this to save the current note back to the database. Good practice is to release any resources that can be released during an `onPause()` as well, to take up less resources when in the passive state. `onResume()` will call our `populateFields()` method to read the note out of the database again and populate the fields.

Why handling life-cycle events is important

If you are used to always having control in your applications, you might not understand why all this life-cycle work is necessary. The reason is that in Android, you are not in control of your Activity, the operating system is!

As we have already seen, the Android model is based around activities calling each other. When one Activity calls another, the current Activity is paused at the very least, and may be killed altogether if the system starts to run low on resources. If this happens, your Activity will have to store enough state to come back up later, preferably in the same state it was in when it was killed.

Activities have a well-defined life cycle. Lifecycle events can happen even if you are not handing off control to another Activity explicitly. For example, perhaps a call comes in to the handset. If this happens, and your Activity is running, it will be swapped out while the call Activity takes over.

So, add some space after the `populateFields()` method and add the following life-cycle methods:

a. `onSaveInstanceState()`:

```
    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        saveState();
        outState.putSerializable(NotesDbAdapter.KEY_ROWID, mRowId);
    }
```

   We'll define `saveState()` next.

b. `onPause()`:

```
    @Override
    protected void onPause() {
        super.onPause();
        saveState();
    }
```

c. `onResume()`:

```
    @Override
    protected void onResume() {
        super.onResume();
        populateFields();
    }
```

Note that `saveState()` must be called in both `onSaveInstanceState()` and `onPause()` to ensure that the data is saved. This is because there is no guarantee that `onSaveInstanceState()` will be called and because when it *is* called, it is called before `onPause()`.

Step 8

Define the `saveState()` method to put the data out to the database.

```
     private void saveState() {
        String title = mTitleText.getText().toString();
        String body = mBodyText.getText().toString();

        if (mRowId == null) {
            long id = mDbHelper.createNote(title, body);
            if (id > 0) {
                mRowId = id;
            }
        } else {
            mDbHelper.updateNote(mRowId, title, body);
        }
    }
```

Note that we capture the return value from `createNote()` and if a valid row ID is returned, we store it in the `mRowId` field so that we can update the note in future rather than create a new one (which otherwise might happen if the life-cycle events are triggered).

Step 9

Now pull out the previous handling code from the `onActivityResult()` method in the `Notepadv3` class.

All of the note retrieval and updating now happens within the `NoteEdit` life cycle, so all the `onActivityResult()` method needs to do is update its view of the data, no other work is necessary. The resulting method should look like this:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    super.onActivityResult(requestCode, resultCode, intent);
    fillData();
}
```

Because the other class now does the work, all this has to do is refresh the data.

Step 10

Also remove the lines which set the title and body from the `onListItemClick()` method (again they are no longer needed, only the `mRowId` is):

```
    Cursor c = mNotesCursor;
    c.moveToPosition(position);
```

and also remove:

```
    i.putExtra(NotesDbAdapter.KEY_TITLE, c.getString(
                    c.getColumnIndex(NotesDbAdapter.KEY_TITLE)));
    i.putExtra(NotesDbAdapter.KEY_BODY, c.getString(
                    c.getColumnIndex(NotesDbAdapter.KEY_BODY)));
```

so that all that should be left in that method is:

```
    super.onListItemClick(l, v, position, id);
    Intent i = new Intent(this, NoteEdit.class);
    i.putExtra(NotesDbAdapter.KEY_ROWID, id);
    startActivityForResult(i, ACTIVITY_EDIT);
```

You can also now remove the mNotesCursor field from the class, and set it back to using a local variable in the `fillData()` method:

```
    Cursor notesCursor = mDbHelper.fetchAllNotes();
```

Note that the `m` in `mNotesCursor` denotes a member field, so when we make `notesCursor` a local variable, we drop the `m`. Remember to rename the other occurrences of `mNotesCursor` in your `fillData()` method.

Run it! (use *Run As -> Android Application* on the project right click menu again)

Solution and Next Steps

You can see the solution to this exercise in `Notepadv3Solution` from the zip file to compare with your own.

When you are ready, move on to the Tutorial Extra Credit exercise, where you can use the Eclipse debugger to examine the life-cycle events as they happen.

← Back to Notepad Tutorial ↑ Go to top