In this exercise, you will add a second Activity to your notepad application, to let the user create and edit notes. You will also allow the user to delete existing notes through a context menu. The new Activity assumes responsibility for creating new notes by collecting user input and packing it into a return Bundle provided by the intent. This exercise demonstrates:

- *Constructing a new Activity and adding it to the Android manifest*
- *Invoking another Activity asynchronously using `startActivityForResult()`*
- *Passing data between Activity in Bundle objects*
- *How to use a more advanced screen layout*
- *How to create a context menu*

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

Step 1

Create a new Android project using the sources from `Notepadv2` under the `NotepadCodeLab` folder, just like you did for the first exercise. 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**.

Open the `Notepadv2` project and take a look around:

- Open and look at the `strings.xml` file under `res/values` — there are several new strings which we will use for our new functionality
- Also, open and take a look at the top of the `Notepadv2` class, you will notice several new constants have been defined along with a new `mNotesCursor` field used to hold the cursor we are using.
- Note also that the `fillData()` method has a few more comments and now uses the new field to store the notes Cursor. The `onCreate()` method is unchanged from the first exercise. Also notice that the member field used to store the notes Cursor is now called `mNotesCursor`. The `m` denotes a member field and is part of the Android coding style standards.
- There are also a couple of new overridden methods (`onCreateContextMenu()`, `onContextItemSelected()`, `onListItemClick()` and `onActivityResult()`) which we will be filling in below.

Step 2

First, let's create the context menu that will allow users to delete individual notes. Open the Notepadv2 class.

1. In order for each list item in the ListView to register for the context menu, we call `registerForContextMenu()` and pass it our ListView. So, at the very end of the `onCreate()` method add this line:

```
registerForContextMenu(getListView());
```

Context menus should always be used when performing actions upon specific elements in the UI. When you register a View to a context menu, the context menu is revealed by performing a "long-click" on the UI component (press and hold the touchscreen or highlight and hold down the selection key for about two seconds).

Because our Activity extends the ListActivity class, `getListView()` will return us the local ListView object for the Activity. Now, each list item in this ListView will activate the context menu.

2. Now fill in the `onCreateContextMenu()` method. This callback is similar to the other menu callback used for the options menu. Here, we add just one line, which will add a menu item to delete a note. Call `menu.add()` like so:

```
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);
    menu.add(0, DELETE_ID, 0, R.string.menu_delete);
}
```

The `onCreateContextMenu()` callback passes some other information in addition to the Menu object, such as the View that has been triggered for the menu and an extra object that may contain additional information about the object selected. However, we don't care about these here, because we only have one kind of object in the Activity that uses context menus. In the next step, we'll handle the menu item selection.

Step 3

Now that the we've registered our ListView for a context menu and defined our context menu item, we need to handle the callback when it is selected. For this, we need to identify the list ID of the selected item, then delete it. So fill in the `onContextItemSelected()` method like this:

```
public boolean onContextItemSelected(MenuItem item) {
    switch(item.getItemId()) {
    case DELETE_ID:
        AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
        mDbHelper.deleteNote(info.id);
        fillData();
        return true;
    }
    return super.onContextItemSelected(item);
}
```

Here, we retrieve the `AdapterContextMenuInfo` with `getMenuInfo()`. The *id* field of this object tells us the position of the item in the ListView. We then pass this to the `deleteNote()` method of our NotesDbAdapter and the note is deleted. That's it for the context menu — notes can now be deleted.

Step 4

Fill in the body of the `createNote()` method:

Create a new `Intent` to create a note (`ACTIVITY_CREATE`) using the `NoteEdit` class. Then fire the Intent using the `startActivityForResult()` method call:

```
Intent i = new Intent(this,
NoteEdit.class);
startActivityForResult(i, ACTIVITY_CREATE);
```

Starting Other Activities

In this example our Intent uses a class name specifically. As well as starting intents in classes we already know about, be they in our own application or another application, we can also create Intents without knowing exactly which application will handle it.

For example, we might want to open a page in a browser, and for this we still use an Intent. But instead of specifying a class to handle it, we use a

This form of the Intent call targets a specific class in our Activity, in this case `NoteEdit`. Since the Intent class will need to communicate with the Android operating system to route requests, we also have to provide a Context (`this`).

predefined Intent constant, and a content URI that describes what we want to do. See `android.content.Intent` for more information.

The `startActivityForResult()` method fires the Intent in a way that causes a method in our Activity to be called when the new Activity is completed. The method in our Activity that receives the callback is called `onActivityResult()` and we will implement it in a later step. The other way to call an Activity is using `startActivity()` but this is a "fire-and-forget" way of calling it — in this manner, our Activity is not informed when the Activity is completed, and there is no way to return result information from the called Activity with `startActivity()`.

Don't worry about the fact that `NoteEdit` doesn't exist yet, we will fix that soon.

Step 5

Fill in the body of the `onListItemClick()` override.

`onListItemClick()` is a callback method that we'll override. It is called when the user selects an item from the list. It is passed four parameters: the `ListView` object it was invoked from, the `View` inside the `ListView` that was clicked on, the `position` in the list that was clicked, and the `mRowId` of the item that was clicked. In this instance we can ignore the first two parameters (we only have one `ListView` it could be), and we ignore the `mRowId` as well. All we are interested in is the `position` that the user selected. We use this to get the data from the correct row, and bundle it up to send to the `NoteEdit` Activity.

In our implementation of the callback, the method creates an `Intent` to edit the note using the `NoteEdit` class. It then adds data into the extras Bundle of the Intent, which will be passed to the called Activity. We use it to pass in the title and body text, and the `mRowId` for the note we are editing. Finally, it will fire the Intent using the `startActivityForResult()` method call. Here's the code that belongs in `onListItemClick()`:

```
super.onListItemClick(l, v, position, id);
Cursor c = mNotesCursor;
c.moveToPosition(position);
Intent i = new Intent(this, NoteEdit.class);
i.putExtra(NotesDbAdapter.KEY_ROWID, id);
i.putExtra(NotesDbAdapter.KEY_TITLE, c.getString(
        c.getColumnIndexOrThrow(NotesDbAdapter.KEY_TITLE)));
i.putExtra(NotesDbAdapter.KEY_BODY, c.getString(
        c.getColumnIndexOrThrow(NotesDbAdapter.KEY_BODY)));
startActivityForResult(i, ACTIVITY_EDIT);
```

- `putExtra()` is the method to add items into the extras Bundle to pass in to intent invocations. Here, we are using the Bundle to pass in the title, body and mRowId of the note we want to edit.
- The details of the note are pulled out from our query Cursor, which we move to the proper position for the element that was selected in the list, with the `moveToPosition()` method.
- With the extras added to the Intent, we invoke the Intent on the `NoteEdit` class by passing `startActivityForResult()` the Intent and the request code. (The request code will be returned to `onActivityResult` as the `requestCode` parameter.)

> **Note:** We assign the mNotesCursor field to a local variable at the start of the method. This is done as an optimization of the Android code. Accessing a local variable is much more efficient than accessing a field in the Dalvik VM, so by doing this we make only one access to the field, and five accesses to the local variable, making the routine much more efficient. It is recommended that you use this optimization when possible.

Step 6

The above `createNote()` and `onListItemClick()` methods use an asynchronous Intent invocation. We need a handler for the callback, so here we fill in the body of the `onActivityResult()`.

`onActivityResult()` is the overridden method which will be called when an Activity returns with a result. (Remember, an Activity will only return a result if launched with `startActivityForResult`.) The parameters provided to the callback are:

- `requestCode` — the original request code specified in the Intent invocation (either `ACTIVITY_CREATE` or `ACTIVITY_EDIT` for us).
- `resultCode` — the result (or error code) of the call, this should be zero if everything was OK, but may have a non-zero code indicating that something failed. There are standard result codes available, and you can also create your own constants to indicate specific problems.
- `intent` — this is an Intent created by the Activity returning results. It can be used to return data in the Intent "extras."

The combination of `startActivityForResult()` and `onActivityResult()` can be thought of as an asynchronous RPC (remote procedure call) and forms the recommended way for an Activity to invoke another and share services.

Here's the code that belongs in your `onActivityResult()`:

```
super.onActivityResult(requestCode, resultCode, intent);
Bundle extras = intent.getExtras();

switch(requestCode) {
case ACTIVITY_CREATE:
    String title = extras.getString(NotesDbAdapter.KEY_TITLE);
    String body = extras.getString(NotesDbAdapter.KEY_BODY);
    mDbHelper.createNote(title, body);
    fillData();
    break;
case ACTIVITY_EDIT:
    Long mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);
    if (mRowId != null) {
        String editTitle = extras.getString(NotesDbAdapter.KEY_TITLE);
        String editBody = extras.getString(NotesDbAdapter.KEY_BODY);
        mDbHelper.updateNote(mRowId, editTitle, editBody);
    }
    fillData();
    break;
}
```

- We are handling both the `ACTIVITY_CREATE` and `ACTIVITY_EDIT` activity results in this method.
- In the case of a create, we pull the title and body from the extras (retrieved from the returned Intent) and use them to create a new note.
- In the case of an edit, we pull the mRowId as well, and use that to update the note in the database.
- `fillData()` at the end ensures everything is up to date .

Step 7

Open the file `note_edit.xml` that has been provided and take a look at it. This is the UI code for the Note Editor.

The Art of Layout

The provided note_edit.xml layout file is the most

This is the most sophisticated UI we have dealt with yet. The file is given to you to avoid problems that may sneak in when typing the code. (The XML is very strict about case sensitivity and structure, mistakes in these are the usual cause of problems with layout.)

There is a new parameter used here that we haven't seen before: `android:layout_weight` (in this case set to use the value 1 in each case).

`layout_weight` is used in LinearLayouts to assign "importance" to Views within the layout. All Views have a default `layout_weight` of zero, meaning they take up only as much room on the screen as they need to be displayed. Assigning a value higher than zero will split up the rest of the available space in the parent View, according to the value of each View's `layout_weight` and its ratio to the overall `layout_weight` specified in the current layout for this and other View elements.

> sophisticated one in the application we will be building, but that doesn't mean it is even close to the kind of sophistication you will be likely to want in real Android applications.
>
> Creating a good UI is part art and part science, and the rest is work. Mastery of XML Layouts is an essential part of creating a good looking Android application.
>
> Take a look at the Hello Views for some example layouts and how to use them. The ApiDemos sample project is also a great resource from which to learn how to create different layouts.

To give an example: let's say we have a text label and two text edit elements in a horizontal row. The label has no `layout_weight` specified, so it takes up the minimum space required to render. If the `layout_weight` of each of the two text edit elements is set to 1, the remaining width in the parent layout will be split equally between them (because we claim they are equally important). If the first one has a `layout_weight` of 1 and the second has a `layout_weight` of 2, then one third of the remaining space will be given to the first, and two thirds to the second (because we claim the second one is more important).

This layout also demonstrates how to nest multiple layouts inside each other to achieve a more complex and pleasant layout. In this example, a horizontal linear layout is nested inside the vertical one to allow the title label and text field to be alongside each other, horizontally.

Step 8

Create a `NoteEdit` class that extends `android.app.Activity`.

This is the first time we will have created an Activity without the Android Eclipse plugin doing it for us. When you do so, the `onCreate()` method is not automatically overridden for you. It is hard to imagine an Activity that doesn't override the `onCreate()` method, so this should be the first thing you do.

1. Right click on the `com.android.demo.notepad2` package in the Package Explorer, and select **New** > **Class** from the popup menu.
2. Fill in `NoteEdit` for the `Name:` field in the dialog.
3. In the `Superclass:` field, enter `android.app.Activity` (you can also just type Activity and hit Ctrl-Space on Windows and Linux or Cmd-Space on the Mac, to invoke code assist and find the right package and class).
4. Click **Finish**.
5. In the resulting `NoteEdit` class, right click in the editor window and select **Source** > **Override/Implement Methods...**
6. Scroll down through the checklist in the dialog until you see `onCreate(Bundle)` — and check the box next to it.
7. Click **OK**.

   The method should now appear in your class.

Step 9

Fill in the body of the `onCreate()` method for `NoteEdit`.

This will set the title of our new Activity to say "Edit Note" (one of the strings defined in `strings.xml`). It will also set the content view to use our `note_edit.xml` layout file. We can then grab handles to the title and body text edit views, and the confirm button, so that our class can use them to set and get the note title and body, and attach an event to the confirm button for when it is pressed by the user.

We can then unbundle the values that were passed in to the Activity with the extras Bundle attached to the calling Intent. We'll use them to pre-populate the title and body text edit views so that the user can edit them. Then we will grab and store the `mRowId` so we can keep track of what note the user is editing.

1. Inside `onCreate()`, set up the layout:

```
setContentView(R.layout.note_edit);
```

2. Change the Activity title to the "Edit Note" string:

```
setTitle(R.string.edit_note);
```

3. Find the `EditText` and `Button` components we need:

   These are found by the IDs associated to them in the R class, and need to be cast to the right type of `View` (`EditText` for the two text views, and `Button` for the confirm button):

```
mTitleText = (EditText) findViewById(R.id.title);
mBodyText = (EditText) findViewById(R.id.body);
Button confirmButton = (Button) findViewById(R.id.confirm);
```

   Note that `mTitleText` and `mBodyText` are member fields (you need to declare them at the top of the class definition).

4. At the top of the class, declare a `Long mRowId` private field to store the current `mRowId` being edited (if any).
5. Continuing inside `onCreate()`, add code to initialize the `title`, `body` and `mRowId` from the extras Bundle in the Intent (if it is present):

```
mRowId = null;
Bundle extras = getIntent().getExtras();
if (extras != null) {
    String title = extras.getString(NotesDbAdapter.KEY_TITLE);
    String body = extras.getString(NotesDbAdapter.KEY_BODY);
    mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);

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

   - We are pulling the `title` and `body` out of the `extras` Bundle that was set from the Intent invocation.
   - We also null-protect the text field setting (i.e., we don't want to set the text fields to null accidentally).

6. Create an `onClickListener()` for the button:

   Listeners can be one of the more confusing aspects of UI implementation, but what we are trying to achieve in this case is simple. We want an `onClick()` method to be called when the user presses the confirm button, and use that to do some work and return the values of the edited note to the Intent caller. We do this using something called an anonymous inner class. This is a bit confusing to look at unless you have seen them before, but all you really need to take away from this is that you can refer to this code in the future to see how to create a listener and attach it to a button. (Listeners are

a common idiom in Java development, particularly for user interfaces.) Here's the empty listener:

```
confirmButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {

    }

});
```

Step 10

Fill in the body of the `onClick()` method of the `OnClickListener` created in the last step.

This is the code that will be run when the user clicks on the confirm button. We want this to grab the title and body text from the edit text fields, and put them into the return Bundle so that they can be passed back to the Activity that invoked this `NoteEdit` Activity. If the operation is an edit rather than a create, we also want to put the `mRowId` into the Bundle so that the `Notepadv2` class can save the changes back to the correct note.

1. Create a `Bundle` and put the title and body text into it using the constants defined in Notepadv2 as keys:

```
Bundle bundle = new Bundle();

bundle.putString(NotesDbAdapter.KEY_TITLE, mTitleText.getText().toString());
bundle.putString(NotesDbAdapter.KEY_BODY, mBodyText.getText().toString());
if (mRowId != null) {
    bundle.putLong(NotesDbAdapter.KEY_ROWID, mRowId);
}
```

2. Set the result information (the Bundle) in a new Intent and finish the Activity:

```
Intent mIntent = new Intent();
mIntent.putExtras(bundle);
setResult(RESULT_OK, mIntent);
finish();
```

   - The Intent is simply our data carrier that carries our Bundle (with the title, body and mRowId).
   - The `setResult()` method is used to set the result code and return Intent to be passed back to the Intent caller. In this case everything worked, so we return RESULT_OK for the result code.
   - The `finish()` call is used to signal that the Activity is done (like a return call). Anything set in the Result will then be returned to the caller, along with execution control.

The full `onCreate()` method (plus supporting class fields) should now look like this:

```
private EditText mTitleText;
private EditText mBodyText;
private Long mRowId;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    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 = null;
    Bundle extras = getIntent().getExtras();
    if (extras != null) {
        String title = extras.getString(NotesDbAdapter.KEY_TITLE);
        String body = extras.getString(NotesDbAdapter.KEY_BODY);
        mRowId = extras.getLong(NotesDbAdapter.KEY_ROWID);

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

    confirmButton.setOnClickListener(new View.OnClickListener() {

        public void onClick(View view) {
            Bundle bundle = new Bundle();

            bundle.putString(NotesDbAdapter.KEY_TITLE,
mTitleText.getText().toString());
            bundle.putString(NotesDbAdapter.KEY_BODY, mBodyText.getText().toString());
            if (mRowId != null) {
                bundle.putLong(NotesDbAdapter.KEY_ROWID, mRowId);
            }

            Intent mIntent = new Intent();
            mIntent.putExtras(bundle);
            setResult(RESULT_OK, mIntent);
            finish();
        }
    });
}
```

Step 11

Finally, the new Activity has to be defined in the manifest file:

Before the new Activity can be seen by Android, it needs its own Activity entry in the `AndroidManifest.xml` file. This is to let the system know that it is there and can be called. We could also specify which IntentFilters the activity implements here, but we are going to skip this for now and just let Android know that the Activity is defined.

There is a Manifest editor included in the Eclipse plugin that makes it much easier to edit the AndroidManifest file, and we will use this. If you prefer to edit the file directly or are not using the Eclipse plugin, see the box at the end for information on how to do this without using the new Manifest editor.

The All-Important Android Manifest File

The AndroidManifest.xml file is the way in which Android sees your application. This file defines the category of the application, where it shows up (or even if it shows up) in the launcher or settings, what activities, services, and content providers it defines, what intents it can receive, and more.

For more information, see the reference document The AndroidManifest.xml File

1. Double click on the `AndroidManifest.xml` file in the package explorer to open it.
2. Click the **Application** tab at the bottom of the Manifest editor.
3. Click **Add...** in the Application Nodes section.

   If you see a dialog with radiobuttons at the top, select the top radio button: "Create a new element at the top level, in Application".
4. Make sure "(A) Activity" is selected in the selection pane of the dialog, and click **OK**.
5. Click on the new "Activity" node, in the Application Nodes section, then type `.NoteEdit` into the *Name** field to the right. Press Return/Enter.

The Android Manifest editor helps you add more complex entries into the AndroidManifest.xml file, have a look around at some of the other options available (but be careful not to select them otherwise they will be added to your Manifest). This editor should help you understand and alter the AndroidManifest.xml file as you move on to more advanced Android applications.

> If you prefer to edit this file directly, simply open the `AndroidManifest.xml` file and look at the source (use the `AndroidManifest.xml` tab in the eclipse editor to see the source code directly). Then edit the file as follows:
> `<activity android:name=".NoteEdit" />`
>
> This should be placed just below the line that reads:
> `</activity>` for the `.Notepadv2` activity.

Step 12

Now Run it!

You should now be able to add real notes from the menu, as well as delete an existing one. Notice that in order to delete, you must first use the directional controls on the device to highlight the note. Furthermore, selecting a note title from the list should bring up the note editor to let you edit it. Press confirm when finished to save the changes back to the database.

Solution and Next Steps

You can see the solution to this exercise in `Notepadv2Solution` from the zip file to compare with your own.

Now try editing a note, and then hitting the back button on the emulator instead of the confirm button (the back button is below the menu button). You will see an error come up. Clearly our application still has some problems. Worse still, if you did make some changes and hit the back button, when you go back into the notepad to look at the note you changed, you will find that all your changes have been lost. In the next exercise we will fix these problems.

Once you are ready, move on to Tutorial Exercise 3 where you will fix the problems with the back button and lost edits by introducing a proper life cycle into the NoteEdit Activity.

← Back to Notepad Tutorial ↑ Go to top