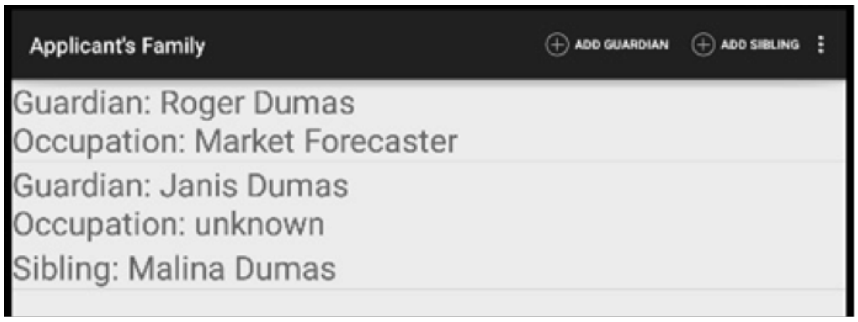
ACTIVITY **2.2.5**

List and Detail

INTRODUCTION

In many applications it is handy to have both list views of your data and views of individual pieces of data. In the case of the CollegeApp, you will create detail views of `Guardian` and `Sibling` and use intents to navigate between them and your list view. This will allow you to start one `Activity` from another. You will also implement an algorithm to prevent users from adding duplicate `FamilyMembers`. Shown below are a list view of the `FamilyMembers` and a detail view of one member, respectively.



Materials

- Computer with Android™ Studio
- Android™ tablet and USB cable, or device emulator
- Free Backendless account per student

RESOURCES

Activity 2.2.5 Visual Aid
Resources available online

Procedure

Part I: Adding and Removing FamilyMembers

1. In Android Studio, open your CollegeApp from *Activity 2.2.4 One Method, Many Classes*. If you were unable to finish the activity, open *`2.2.4CollegeApp_Solution`* as directed by your teacher. (If you use the solution code, change `BE_APP_ID, BE_ANDROID_API_KEY`, and `MY_EMAIL_ADRESS` in *`ApplicantActivity.java`* to reference your personal Backendless and email values.)

You will now add a context menu to allow the user to remove members from their family, and an action bar menu to allow the user to add members to their family.

2. To create the context menu, first create a new xml file in the *`res/menu`* folder. Call it *`family_list_item_context.xml`*. Do not change any of the other default settings.
3. Within the xml file that you created in the previous step, add lines 2–4 below.

```
1: <menu xmlns:android="http://schemas.android.com/apk/res/android">
2: <item android:id="@+id/menu_item_delete_family_member"
3:     android:icon="@android:drawable/ic_menu_delete"
4:     android:title="@string/delete_family_member" />
5: </menu>
```

4. Add a string resource for your new string in the appropriate location with the value `"Delete Family Member"`.
5. Create another menu layout file named *`fragment_family_list.xml`*.
6. Review the slideshow.

PROJECT LEAD THE WAY PLTW

2.2.5 List and Detail

Computer Science A © 2016 Project Lead The Way, Inc.

The methods above are all important as you attempt to set up an action bar menu and a context menu in this activity.

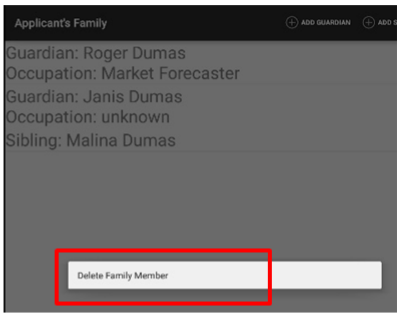
At top right a context menu has been outlined. The context menu was activated by long-pressing on an item in the list.

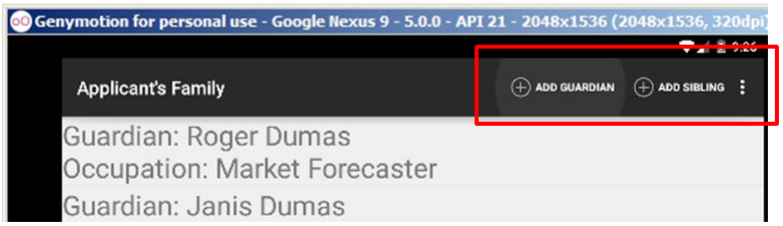
At bottom, three items in an action bar menu have been highlighted. In the screenshot, ADD GUARDIAN is being tapped, indicated by the slightly different coloration on that item. At the far right, the third item is the vertically oriented ellipses which, when tapped, expands into additional menu items. All three of these items are part of the OptionsMenu.

None of the menu specific methods or knowledge are part of the AP subset

Menus

- Relevant methods:
  `onCreate`
  `onCreateView`
  `onCreateOptionsMenu`
  `onOptionsItemSelected`
  `onCreateContextMenu`
  `onContextItemSelected`





Menus: `onCreate` and `onCreateView`

In the `onCreate` method:

```
setHasOptionsMenu(true);
```

In the `onCreateView` method:

```
ListView listView =
   (ListView)v.findViewById(android.R.id.list);

registerForContextMenu(listView);
```


The first statement belongs in onCreate and tells your activity or fragment that it will have an options menu in the action bar.

The second statement belongs in onCreateView and creates a new ListView object using the R resource file.

The third statement follows the second, and tells the activity or fragment that it will contain a context menu for a given UI element. In this case, for the ListView.

Menus: `onCreateOptionsMenu`

```
@Override
   public void onCreateOptionsMenu(Menu menu,
                                   MenuInflater inflater){

       super.onCreateOptionsMenu(menu, inflater);

       inflater.inflate(R.menu.fragment_family_list, menu);
    }
```

Computer Science A

The onCreateOptionsMenu should inflate a layout element, a menu, in this case R.menu.fragment_family_list. This layout file contains the layout for the menu that will appear in the action bar of FamilyListFragment.

Inflate has the same meaning as build or create. The inflater is going to bring the menu you defined in the XML file into existence in the app.

Menus: `onOptionsItemSelected`

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {

      FamilyMemberAdapter adapter =
                      (FamilyMemberAdapter)getListAdapter();

      switch (item.getItemId()) {
      case R.id.menu_item_new_guardian:
            … implementation here …

      case R.id.menu_item_new_sibling:
            … implementation here …

      default:
            return super.onOptionsItemSelected(item);
        }
    }
```

Computer Science A

In onOptionsItemSelected, an instance of FamilyMemberAdapter is created so that you can notify the list of family members of updates, like when a new Guardian or Sibling is added.

This method uses a switch case sequence to decide which kind of FamilyMember was clicked, (not in the AP Subset) but you could just as easily use an if – else if – else sequence.

The parameter, item, at run time will be a representation of the item that was clicked in the action bar menu, and the getItemId method will return the id of that item so that you can tell which set of operations to complete.

Menus: `onCreateContextMenu`

```
@Override
   public void onCreateContextMenu(
                   ContextMenu menu,
                   View v,
                   ContextMenu.ContextMenuInfo menuInfo) {


      Log.d(TAG, "Creating Context Menu.");

      getActivity().
           getMenuInflater().
                 inflate(R.menu.family_list_item_context, menu);
   }
```

Computer Science A

This method only inflates the family list item context menu layout.

Menus: `onContextItemSelected`

```
@Override
public boolean onContextItemSelected(MenuItem item) {

    AdapterView.AdapterContextMenuInfo info =
    (AdapterView.AdapterContextMenuInfo)item.getMenuInfo();

    int position = info.position;

    FamilyMemberAdapter adapter =

(FamilyMemberAdapter)getListAdapter();
    FamilyMember familyMember = adapter.getItem(position);

    switch (item.getItemId()) {
    case R.id.menu_item_delete_family_member:
       … implementation detail …

    }
        return super.onContextItemSelected(item);
    }
```

Computer Science A

This method connects into the adapter for the family member list and finds the position of the item that was long-clicked. Since this context menu will have only one operation, delete, there is only one case in the switch-case sequence.

If you don't understand many of the statements here, that's ok, the important part here would be to be able to add more cases if you wanted to and fill in the part hidden with "… implementation detail…"

7. Within the xml file that you created in the previous step, delete the closing angle bracket at the end of your line 2, and add lines 2–10 below.

```
 1: <menu xmlns:android="http://schemas.android.com/apk/res/
    android"
 2:       xmlns:app="http://schemas.android.com/apk/res-auto">
 3:     <item android:id="@+id/menu_item_new_guardian"
 4:           android:icon="@android:drawable/ic_menu_add"
 5:           android:title="@string/new_guardian"
 6:           app:showAsAction="ifRoom|withText"/>
 7:     <item android:id="@+id/menu_item_new_sibling"
 8:           android:icon="@android:drawable/ic_menu_add"
 9:           android:title="@string/new_sibling"
10:           app:showAsAction="ifRoom|withText"/>
11: </menu>
```

8. Add the relevant string values, `"Add Guardian"` and `"Add Sibling"` respectively.
9. Add the following six methods to `FamilyListFragment`.
10. Review the slideshow.

Intents

- Use Intent class objects to start a new Activity
- You can add data to send to the new Activity

```
Intent i = new Intent(getActivity(),
                      FamilyMemberActivity.class);
i.putExtra(FamilyMember.EXTRA_INDEX, position);
```

- Start a new activity from the current Fragment or Activity

```
startActivity(i);
```

- Get information from the Intent that started your Activity

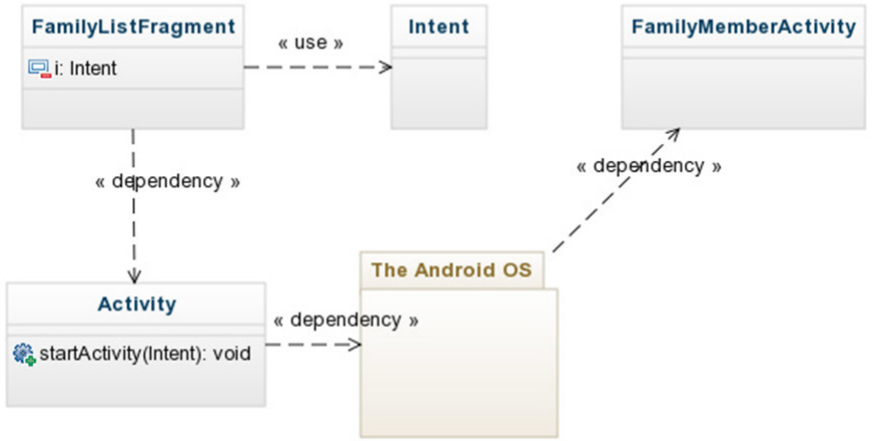
```
getIntent().getIntExtra(FamilyMember.EXTRA_INDEX, 0);
```

Computer Science A

For more information on starting Activities and using Intents:

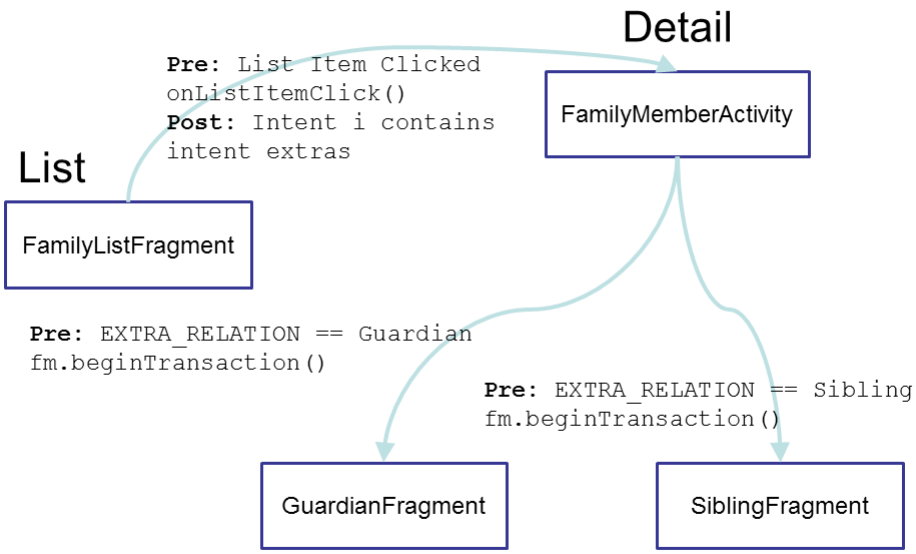
http://developer.android.com/training/basics/firstapp/starting-activity.html

Intents Cont.



Computer Science A

How List and Detail Fit Together



Computer Science A

Upon a click on a list item in FamilyListFragment, a call is made to onListItemClick, which creates an Intent, and puts a couple of extras in it including EXTRA_RELATION and then calls startActivity with that intent.

Based on the value of the intent extra passed to FamilyMemberActivity, it will either start a fragment manager transaction to add a GuardianFragment or a SiblingFragment as appropriate

```
 1: @Override
 2: public View onCreateView(LayoutInflater inflater, ViewGroup parent,
 3:                          Bundle savedInstanceState) {
 4:     View v = super.onCreateView(inflater, parent,
                                    savedInstanceState);
 5:
 6:     ListView listView = (ListView)v.findViewById(android.R.id.
                                                        list);
 7:     registerForContextMenu(listView);
 8:
 9:     return v;
10: }
11:
12: @Override
13: public void onCreateOptionsMenu(Menu menu, MenuInflater
    inflater) {
14:     super.onCreateOptionsMenu(menu, inflater);
15:     inflater.inflate(R.menu.fragment_family_list, menu);
16: }
17:
18: @Override
19: public boolean onOptionsItemSelected(MenuItem item) {
20:      FamilyMemberAdapter adapter = (FamilyMemberAdapter)
         getListAdapter();
21:      switch (item.getItemId()) {
22:          case R.id.menu_item_new_guardian:
23:              Log.d(TAG, "Selected add new guardian.");
24:              Guardian guardian = new Guardian();
25:              Family.get().addFamilyMember(guardian);
26:              adapter.notifyDataSetChanged();
27:              return true;
28:          case R.id.menu_item_new_sibling:
29:              Log.d(TAG, "Selected add new sibling.");
30:              Sibling sibling = new Sibling();
31:              Family.get().addFamilyMember(sibling);
32:               adapter.notifyDataSetChanged();
33:              return true;
34:          default:
35:              return super.onOptionsItemSelected(item);
36:      }
37: }
38:
39: @Override
40: public void onCreateContextMenu(ContextMenu menu, View v,
41:    ContextMenu.ContextMenuInfo menuInfo) {
42:      Log.d(TAG, "Creating Context Menu.");
43:      getActivity().getMenuInflater().inflate(R.menu.family_list_
44:      item_context, menu);
```

```
45: }
46:
47: @Override
48: public boolean onContextItemSelected(MenuItem item) {
49:    Log.d(TAG, "Context item selected.");
50:    AdapterView.AdapterContextMenuInfo info =
51:            (AdapterView.AdapterContextMenuInfo) item.
            getMenuInfo();
52:    int position = info.position;
53:    FamilyMemberAdapter adapter = (FamilyMemberAdapter)
       getListAdapter();
54:    final FamilyMember familyMember = adapter.getItem(position);
55:
56:    switch (item.getItemId()) {
57:        case R.id.menu_item_delete_family_member:
58:            Family.get().deleteFamilyMember(familyMember);
59:            adapter.notifyDataSetChanged();
60:            Backendless.Data.of(FamilyMember.class).
               remove(familyMember,new
61:            AsyncCallback<Long>() {
62:                @Override
63:                public void handleResponse(Long response) {
64:                    Log.i(TAG, familyMember.toString() + "
                       deleted");
65:                }
66:
67:                @Override
68:                public void handleFault(BackendlessFault fault)
                   {
69:                    Log.e(TAG, fault.getMessage());
70:                }
71:            });
72:            return true;
73:            }
74:     return super.onContextItemSelected(item);
75: }
76:
77: @Override
78: public void onResume() {
79:     super.onResume();
80:     FamilyMemberAdapter adapter = (FamilyMemberAdapter)
        getListAdapter();
81:     adapter.notifyDataSetChanged();
82: }
```

This will show errors related to the calls `addFamilyMember` and `deleteFamilyMember`.

11. To get rid of the errors, create `addFamilyMember` and `deleteFamilyMember` methods for the `Family` class. Implement the addition and removal of `family` members. Because the family object is an `ArrayList`, `addFamilyMember` can call `family.add()`, and `deleteFamilyMember` can call `family.remove()`. Note that both of these calls require the `FamilyMember` parameter that is passed to each method.

12. To indicate that `FamilyListFragment` now contains custom items in its Options menu, add line 2 shown below to its `onCreate` method.

```
1:     setListAdapter(adapter);
2:     setHasOptionsMenu(true);
3: }
```

13. Test your app. You should see the new menu items and be able to add new guardians and siblings.

Part II: Comparing Objects Using == and Equals

When running your app, notice how you can add multiple siblings and guardians with the same name. Ideally, you want to add only unique entries to the list of family members.

14. To get a start on that, in `FamilyListFragment` in your new `onOptionsItemSelected` method, before you add a guardian with `Family.get().addFamilyMember(guardian);`, write a for-each loop:

```
1: for (FamilyMember f: Family.get().getFamily()) {
2: }
```

15. To test whether two objects are the same, experiment with some comparison algorithms. Add the following lines inside the for loop that you created in the previous step:

```
1: if (f == guardian) {
2:     Log.i(TAG, "Possible match " + guardian + " and" + f);
3: }
```

16. Run the app, add a few guardians, and report how many duplicate entries the app finds.

    Is this what you predicted? Recall from activity 1.1.4 If It's Raining... the use of == versus the `equals()` method. What are you comparing?

    To correctly compare two objects in your app and avoid duplicating family members, you will create an `equals` method in `Guardian`, to override the `String's` equals method.

17. Modify `Guardian` so that it contains an `equals` method.

    This method should return `true` when the first name and the last name of the `Guardian` parameter are the same as this `Guardian's` first and last name; otherwise the method should return `false`.

18. To ensure siblings are not duplicated, modify `Sibling` so that it contains an `equals` method.

19. Notice that identical code now exists in `Sibling` and `Guardian`.

    What might be a better solution?

    > **Hint**: What do they both extend?

    Check your answer

    It would be better to put the code in the FamilyMember class, since Sibling and Guardian are both subclasses of FamilyMember and could inherit the method.

20. Implement a single `equals` method in the parent class so that both `Guardian` and `Sibling` classes can use it. Make sure that this method checks to see whether the objects it is comparing are really *instances of* the same class. The code for a guardian is partially implemented below; you still need to implement the call to `equals` to compare first and last names

```
public boolean equals(Object o) {
    if ((o instanceof Guardian) && (this instanceof Guardian)) {
        // both are guardians so cast the Object
        Guardian g = (Guardian)o;
        // test for equality of first and last names for g and
        this
```

21. In an `else` if, test for equality of siblings.

22. Now that the parent class does the comparison for equality, remove the unnecessary `equals` methods in both Guardian and Sibling classes.

23. Back in `FamilyListFragment` in the `onOptionsItemSelected` method, change the conditional statement (that you added earlier in this part of the activity), so you can recognize family members with duplicate names. Only add a new family member, if one does not already exist with the same name.

    As you write the iteration algorithm, know that one of the restrictions of the for loop you are using is that you cannot modify the Singleton Family within the iteration. (If you attempt this, your app will throw a `ConcurrentModificationException`.) You must iterate over all of

the family members first, to determine whether a duplicate exists. So a good solution would be to create a `boolean` variable that you can set if a duplicate name is found. After you search *all* family members and a duplicate is not found, you can add a new guardian.

24 Write a similar algorithm to ensure that siblings are not duplicated.

An interesting side effect has occurred; your app cannot have a sibling and a guardian with the same name. Is this ideal? Why or why not?

Part III: List View, Detail View

After you complete this part of the activity, you will be able to switch between a list view of `FamilyMembers` and views of individual `Guardians` and `Siblings`.

Currently the app does nothing when you tap on a `FamilyMember` view in the list. In this part of the activity, you will implement functionality by starting a `FamilyMemberActivity` detail view of your `FamilyMember` objects when they are tapped.

25 Create a layout file for this class named *`activity_family_member.xml`*. Use `LinearLayout` as the root element and place the following `FrameLayout` within it.

```
1: <FrameLayout xmlns:android="http://schemas.android.com/apk/
   res/android"
2:                    android:id="@+id/fragmentContainer"
3:                    android:layout_width="match_parent"
4:                    android:layout_height="match_parent"/>
```

26 Add the following method to `FamilyListFragment`.

```
1: @Override
2: public void onListItemClick(ListView l, View v, int
                               position, long id) {
3:      FamilyMember f = ((FamilyMemberAdapter)
        getListAdapter()).  getItem(position);
4:      Log.d(TAG, f.toString() + " was clicked." +
        FamilyMemberActivity.class);
```

```
5:      Intent i = new Intent(getActivity(),
        FamilyMemberActivity.class);
6:      i.putExtra(FamilyMember.EXTRA_RELATION, f.getClass().
        getName());
7:      i.putExtra(FamilyMember.EXTRA_INDEX, position);
8:      startActivity(i);
9: }
```

This will generate some errors, which you will fix in the next few steps.

Use your resources to identify what each of the parameters of this method do, and record their functionality in your own words.

27. Add the `String` constants to `FamilyMember` that are used in the `onListItemClick` method. Use the values `"org.pltw.examples.collegeapp.relation"` and `"org.pltw.examples.collegeapp.index"`.
28. Now create a `FamilyMemberActivity` class that extends `FragmentActivity`. This class will host a different `Fragment` depending on whether the list item click that started it targeted a `Guardian` or a `Sibling`.
29. Add to `FamilyMemberActivity` the following method.

```
 1: @Override
 2: public void onCreate(Bundle savedInstanceState) {
 3:
 4:     super.onCreate(savedInstanceState);
 5:     setContentView(R.layout.activity_family_member);
 6:     FragmentManager fm = getSupportFragmentManager();
 7:     Fragment fragment = null;// = fm.findFragmentById(R.
        id.fragmentContainer);
 8:
 9:     if (fragment == null) {
10:         if (getIntent().getStringExtra(FamilyMember.EXTRA_
            RELATION).equals(Guardian.class.getName())) {
11:             fragment = new GuardianFragment();
12:             fm.beginTransaction()
13:                     .add(R.id.fragmentContainer, fragment)
14:                     .commit();
15:         }
16:         else if (getIntent().getStringExtra(FamilyMember.
            EXTRA_RELATION).equals(Sibling.class.getName())) {
17:             fragment = new SiblingFragment();
18:             fm.beginTransaction()
19:                     .add(R.id.fragmentContainer, fragment)
20:                     .commit();
21:         }
22:     }
23: }
```

Now you will create separate fragments for your guardian and sibling family members.

30 First, change (refactor) *fragment_family_member.xml* to *fragment_guardian.xml* to better reflect the purpose of the resource file.

a. In *fragment_guardian.xml*, change (refactor) ids so all references to "familyMember" are replaced with "guardian".

b. Continuing in *fragment_guardian.xml*, create a `TextView` and `EditText` for occupation.

In Android development, developers refer to “wiring up” widgets. In fact, you have been wiring up many widgets in CollegeApp already. To `wire up` a widget means to:

- Get the reference id of a widget (such as a `TextView`) using the `findViewById` method. The ids are usually defined in XML resource files.
- Define the event handler or listener for the widget, so that it responds to user actions.

31 Create a `SiblingFragment` class that extends `Fragment`.

a. Create a new layout file for your `SiblingFragment` using *fragment_guardian.xml* as a model (do not include an occupation). Check that your ids refer to "sibling" and not "guardian".

b. Wire up your new fragment based on `GuardianFragment`.

32 If you try to run your app now, it will crash when you attempt to select a `FamilyMember`. The log will show you a message asking, "Have you declared this activity in your AndroidManifest.xml?" This means that you have not yet added `FamilyMemberActivity` to the Android manifest.

33 In *AndroidManifest.xml*, add lines 2–4 below.

```
1:         </activity>
2:         <activity android:name=".FamilyMemberActivity"
3:                   android:label="CollegeApp">
4:         </activity>
5:
6:     </application>
7: </manifest>
```

34 Test your code.

NOTE

When you navigate from the “Applicant’s Family” fragment to the “Family Member Content” fragment, you no longer have a NavDrawer for navigation. Instead, use the Back button on your device to navigate back to your FamilyList fragment.

35 Improve your code.

Before your views will display all applicant and family data correctly from the database, you need to modify the persistence of your data. Modify your code to achieve the following objectives. Your teacher will tell you how many of these to implement.

As you make these changes to CollegeApp, you should occasionally delete all family members and create new ones to properly test new functionality.

a. Modify `ProfileFragment` so that using the Submit button saves the applicant's data in the database and navigating away does not save. The `onPause` method will no longer save the data, only the Submit button will.

b. Clicking on a `Guardian` in `FamilyListFragment` starts a `FamilyMemberActivity` that displays the correct `Guardian's` information.

   i. Override the `onStart` method in `GuardianFragment`.

   ii. Access your `Family` information based on the guardian's *position* in the `Family` list. This position was put on the Intent in `FamilyListFragment's onListItemClick` using the Intent data associated with `EXTRA_INDEX.` As a review for how to get this data *off* of the intent, refer to `FamilyMemberActivity's onCreate` method. To get an integer value from the intent, use:

```
index = getActivity().getIntent().getIntExtra(FamilyMember.
EXTRA_INDEX, -1);
```

   iii. If the index is not –1, get the element in `Family` array list and assign it to `mGuardian`. You will have to cast the family member to the type `Guardian`.

   iv. Finally, set the appropriate `TextViews` with the values of `mGuardian`.

c. Use similar algorithms so that clicking on a `Sibling` in `FamilyListFragment` starts a `FamilyMemberActivity` that displays the correct `Sibling` information.

d. Modify `GuardianFragment` so that using the Submit button saves to the database.

   i. Modify the Submit button's `setOnClickListener`.

   ii. Save the current `mGuardian` to Backendless.

   iii. Add functionality so that the guardian in Backendless has your email address; otherwise you will not be able to retrieve it. Hint: when you create a `new Guardian` in `FamilyListFragment`, set its email. You will need to retrieve it from shared preferences. The email addresses will be added to the *new* guardians you add to your list.

e. Use similar algorithms for `SiblingFragment` so that tapping the Submit button saves to the database for new siblings you add to your list.

f. When `FamilyListFragment` starts (hint: `onStart()`), it displays all guardians and siblings from the database instead of the ones you created in using your `Family` class.

g. You may have noticed that a family member is not saved until you modify it. Add a new save feature to the list in `FamilyListFragment`.

   i. Add a save menu item (similar to delete menu item) to *`family_list_item_context.xml`*, including the string value in *strings.xml*.

   ii. In `onContextItemSelected`, add a new case `R.id.menu_item_save_family_member`: and save the guardian and sibling family members in the database.

36 Test your app. You should be able to navigate between a list view of the applicant's family and detail views of individual members, save family member information, and update the applicant's profile data.

Part IV: Dangerous Permission

In addition to defining an app's activities, the manifest file is responsible for defining permissions that an app may need. These permissions can include accessing private data on the device or sharing the location of the device.

37 Read the following excerpt from the Android Developers Guide regarding **Normal and Dangerous Permissions**:

> "Normal permissions cover areas where your app needs to access data or resources outside the app's sandbox, but where there's very little risk to the user's privacy or the operation of other apps. For example, permission to set the time zone is a normal permission. If an app declares that it needs a normal permission, the system automatically grants the permission to the app.
>
> Dangerous permissions cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps. For example, the ability to read the user's contacts is a dangerous permission. If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app."

38 Also from **Normal and Dangerous Permissions**, review the types of permissions that the Android operating system considers dangerous:

Permission Group	Permissions
CALENDAR	READ_CALENDAR WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

Which do you think is the most “dangerous permission” and why?

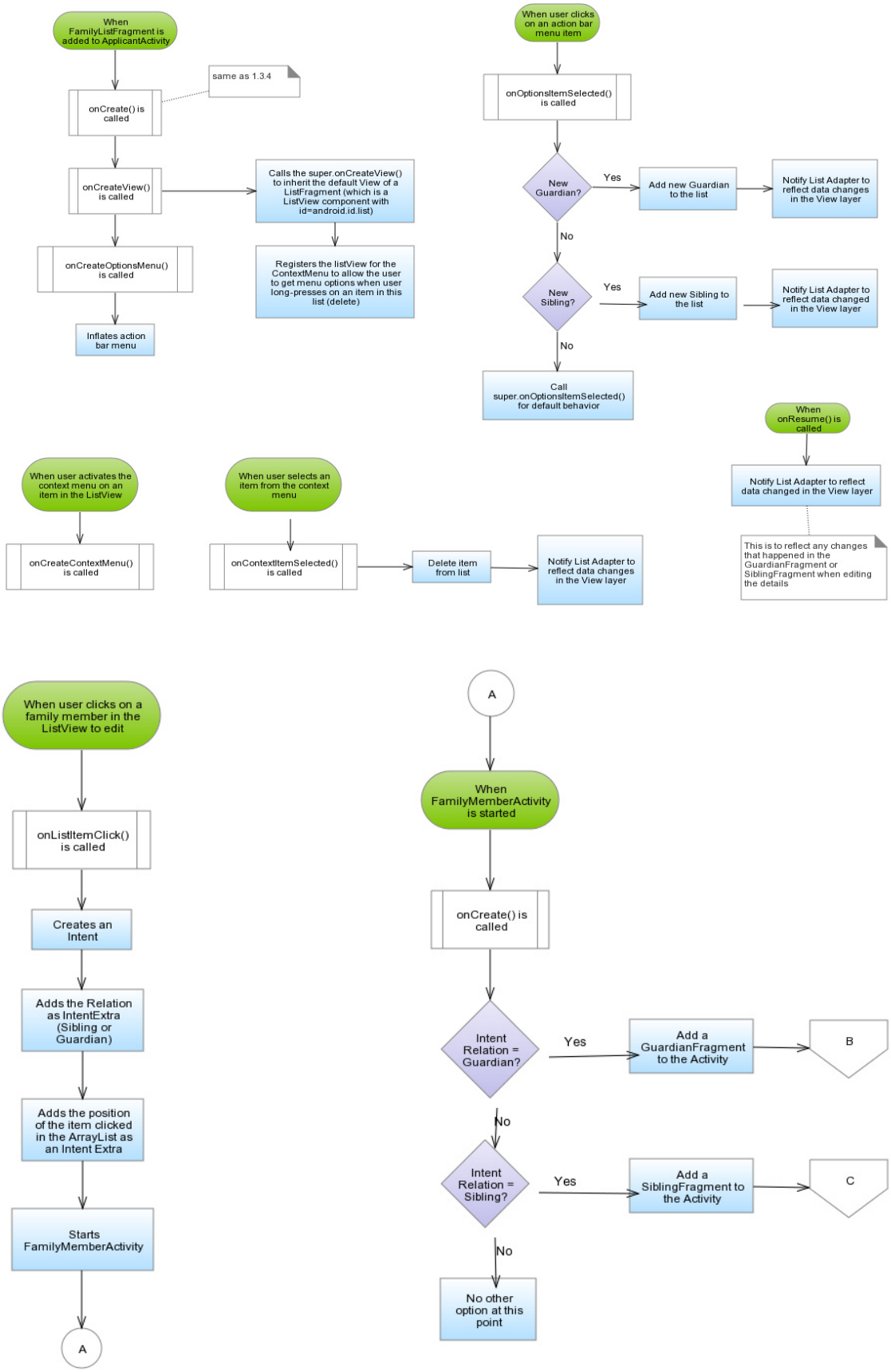
By default, an app does not have any of these permissions, but it could potentially have *all* of them.

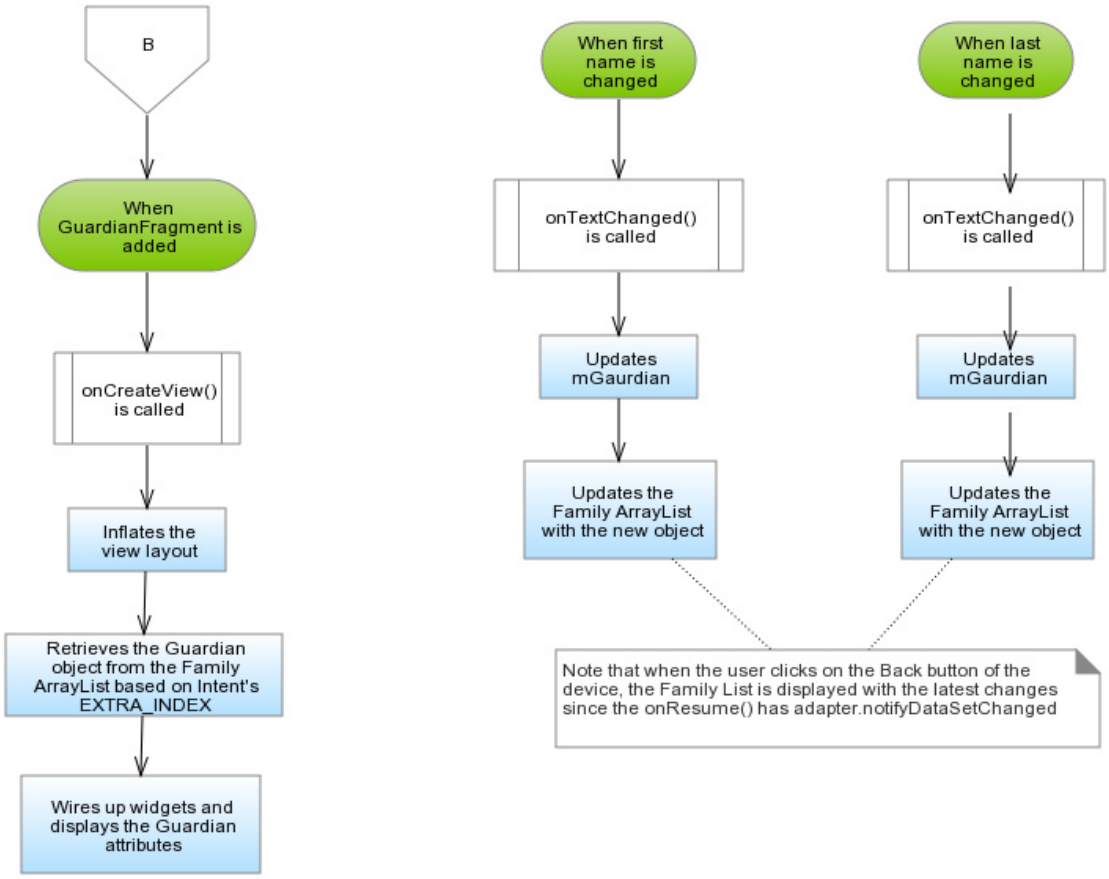
If an app uses permissions from the Phone and SMS groups, why would it be *irresponsible* for a programmer to include all permission groups in the Android manifest file? Why might a programmer choose to include all?

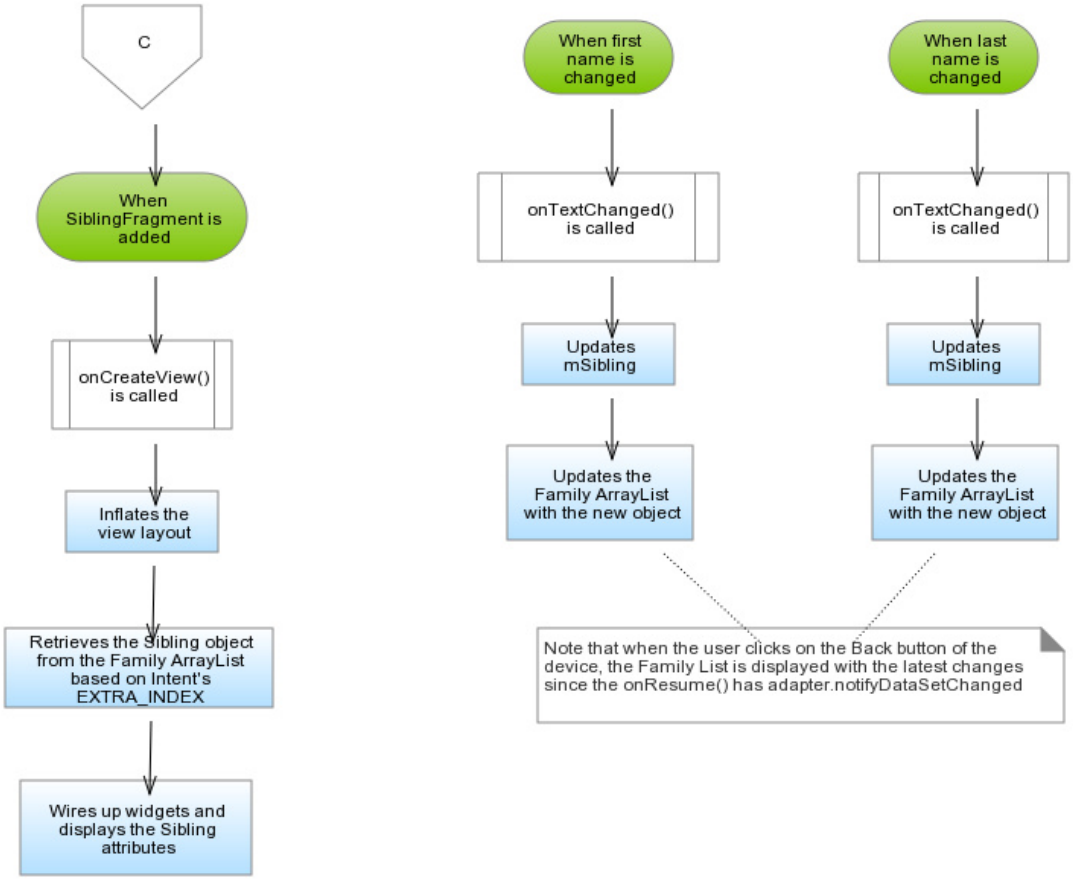
CONCLUSION

1. In your own words, explain how intents fit into the Android app lifecycle.
2. What do you think is the least secure or most dangerous part of this app for your users?

Activity 2.2.5 Visual Aid




B
When GuardianFragment is added
onCreateView() is called
Inflates the view layout
Retrieves the Guardian object from the Family ArrayList based on Intent's EXTRA_INDEX
Wires up widgets and displays the Guardian attributes
When first name is changed
onTextChanged() is called
Updates mGaurdian
Updates the Family ArrayList with the new object
When last name is changed
onTextChanged() is called
Updates mGaurdian
Updates the Family ArrayList with the new object
Note that when the user clicks on the Back button of the device, the Family List is displayed with the latest changes since the onResume() has adapter.notifyDataSetChanged


C
When SiblingFragment is added
onCreateView() is called
Inflates the view layout
Retrieves the Sibling object from the Family ArrayList based on Intent's EXTRA_INDEX
Wires up widgets and displays the Sibling attributes
When first name is changed
onTextChanged() is called
Updates mSibling
Updates the Family ArrayList with the new object
When last name is changed
onTextChanged() is called
Updates mSibling
Updates the Family ArrayList with the new object
Note that when the user clicks on the Back button of the device, the Family List is displayed with the latest changes since the onResume() has adapter.notifyDataSetChanged