PROJECT **2.1.5**

User Input

INTRODUCTION

One of the most basic qualities of an effective app is its ability to gather input from the user. Unlike desktop applications, mobile apps register a variety of gestures on their touch screens as well as data from other onboard sensors. This makes designing for mobile devices significantly different from designing for conventional personal computers. What kind of input have you provided to a mobile device?

In this project, you will extend the College App to accept input when the user clicks `Family Member` or `Profile` options in the Navigation Drawer.

Materials

- Computer with Android™ Studio
- Android™ tablet and USB cable, or a device emulator

RESOURCES

Optional Extension: Create a Date Picker Dialog
Resources available online

College App Problem Statement
Resources available online

Procedure

Part I: Viewing the Model Data

1. Form pairs as directed by your instructor.
2. Greet your partner and set team norms for pair programming.

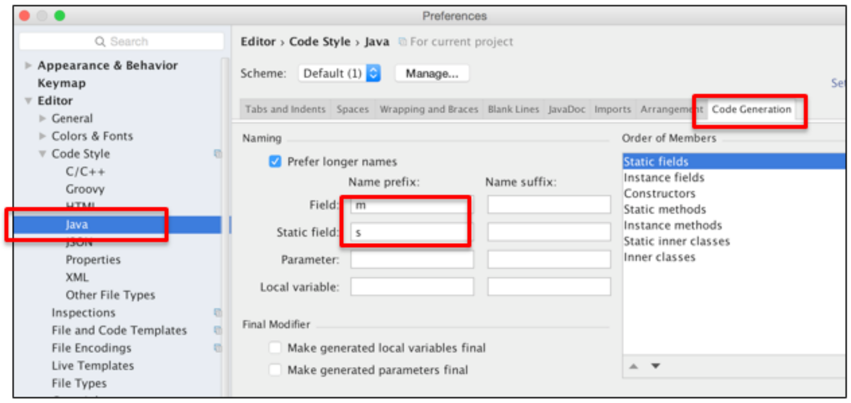
You will now implement Model layer classes for the College App: `Profile` to store information about the applicant, and `FamilyMember` to store information about one of the applicant's family members.

3. Open your copy of College App; if you were unable to complete *Activity 2.1.4 App Navigation*, obtain and import `2.1.4CollegeApp_Solution` as directed by your teacher.
4. Create a new class named `FamilyMember`.
5. Give this class two private instance variables: `firstName` and `lastName`.
6. To add accessor and mutator methods for each of these variables, right-click an empty line in the `FamilyMember` class and select **Generate...** > **Getter and Setter**. Highlight both of your variables and select **OK**.

Now that you have created a Model layer class for your app, you will wire it to your Presenter and View layers. In Android development, it is common practice to precede instance fields, also called member fields, with the prefix “m”. This is called **Hungarian naming convention**. Its advantage is that it reduces naming collisions with parameters of methods. You’ll likely see this convention in many official documents that you come across.

7. Within `FamilyMemberFragment`, create `TextView` and `EditText` instance fields for first name and last name using the Hungarian naming convention that you just learned.
    - `TextViews` are the Java objects that handle display of text in Android apps.
    - `EditTexts` are the Java objects that allow a user to enter new text.

    Hungarian naming can have an unintended consequence when you use the getter and setter generating feature of Android Studio. For example, if you tried to automatically generate a getter for `mLastName`, you would get a method named `getmLastName`, which sounds more awkward than `getLastName`.
8. To avoid this, you can change the settings for auto-generated code in Android Studio. An example of these sub-steps are show below.
    a. Open the Preferences window by going to **File** > **Settings** (or **Android Studio** > **Preferences** if you are on a Mac). From the panel on the left side, click **Editor** to expand it**,** then click **Code Style** to expand it, and then click **Java**.
    b. Click the **Code Generation** tab on the far right, and change the values of Name prefix for Field and Static field to “m” and “s”, respectively.
    c. Click **OK** when you are done.



Editor › Code Style › Java For current project

Scheme: Default (1)

Naming

[✓] Prefer longer names

	Name prefix:	Name suffix:
Field:	m	
Static field:	s	
Parameter:		
Local variable:		

Final Modifier

[ ] Make generated local variables final

[ ] Make generated parameters final

Order of Members

Static fields
Instance fields
Constructors
Static methods
Instance methods
Static inner classes
Inner classes

9 Create another instance field for `FamilyMemberFragment` that is of type `FamilyMember`.

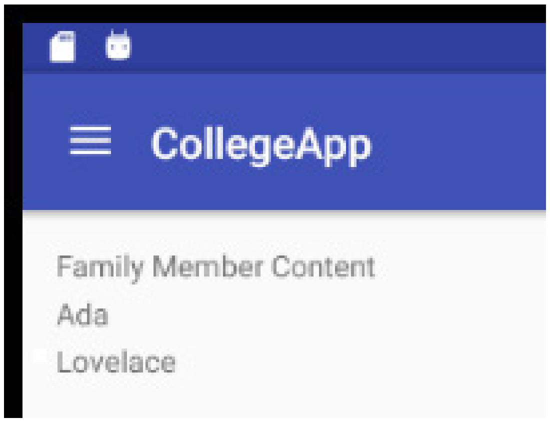
10 In your `FamilyMember` class, initialize the first and last names of your `FamilyMember` with a new constructor:

```
public FamilyMember() {
     firstName = "Ada";
     lastName = "Lovelace";
}
```

11 Using the skills that you acquired in problem *1.2.4 Create an Android Project*, modify any Model, View, and Presenter code so that the newly initialized values of your `FamilyMember` will show up in the app.

Remember that you will need to invoke `findViewById` on `rootView` to get the layout elements from within your `Fragment`. You will also need to use `TextView`'s `setText` method.

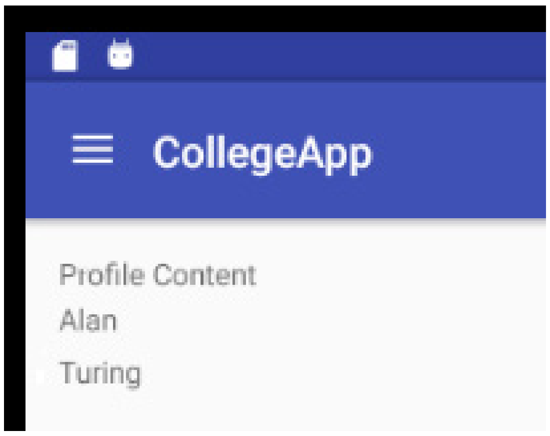
Your finished `FamilyMemberFragment` should look like this:



Family Member Content
Ada
Lovelace

12 As you did for `FamilyMember`, initialize the first and last names of your `Profile` with a new constructor.

13 Create a Model layer class for the applicant's `Profile` and wire it up as you did the `FamilyMember`. After doing this, your `ProfileFragment` should look like this:



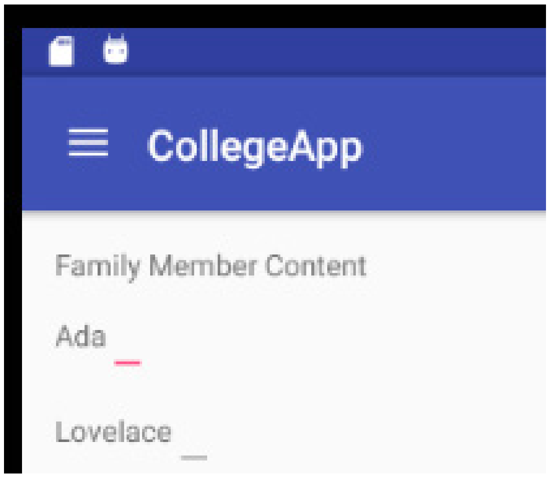
Profile Content
Alan
Turing

Part II: User Input: Text Fields

`EditText` is the Java® object type associated with text fields in Android development.

14 Create two `EditText` objects for the first and last names of your `FamilyMemberFragment`.

15 In the corresponding XML file, add `EditText` elements (called "Plain Text" in the graphical layout editor) and any other layout elements you need to achieve a layout like the following:



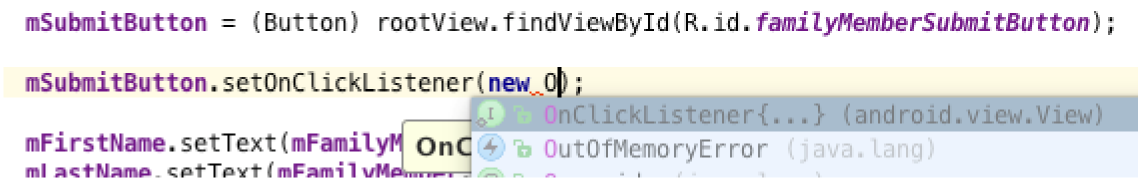
16 Wire the Presenter layer for this new `EditText` to the View layer.

17 Add a `Button` named `mSubmitButton` to the `FamilyMemberFragment` so that when the user taps this new button, it updates the first and last names by replacing them with the contents of the text field.

Previously you used the `onClick` XML property to do this. When working with Fragments, it is much more convenient to use anonymous inner classes. The following steps will help you set up a listener for your button's click using an anonymous inner class.

a. Call the `setOnClickListener` method of `mSubmitButton`.

b. As a parameter, type `new O` and select the first autocomplete option that appears:



```
mSubmitButton = (Button) rootView.findViewById(R.id.familyMemberSubmitButton);

mSubmitButton.setOnClickListener(new O);

mFirstName.setText(mFamilyM
mLastName.setText(mFamilyMe
```

OnC
OnClickListener{...} (android.view.View)
OutOfMemoryError (java.lang)

This automatically generates the `OnClickListener` that you need, to add functionality to your `Button's` click.

c. Fill in the `onClick` method that appears with the code that you want to execute when the `Button` is clicked.

18. Test your button to make sure that the Model and Presenter layer objects are being changed as you expect.

19. Once they are, test what happens if you change the first or last name, and then navigate away from the `FamilyMemberFragment` and then back to it. Do the names remain changed? Add similar functionality to your `ProfileFragment` and test your app for both profile and family member functionality.

Part III: Abstract Classes

`FamilyMember` isn't a very specific class of people, and it doesn't make much sense to implement as a class it if there are not enough details. Two types of people in a college applicant's life are likely to matter on a college application: parents/guardians and siblings. In this part of the project, you will implement subclasses of the `FamilyMember` superclass.

In this app we will implement these types of family members as `Guardian` and `Sibling` classes. What data fields do you imagine each having?

Guardian	Sibling

Both `Guardians` and `Siblings` are types of `FamilyMembers;` they will inherit fields and methods from `FamilyMember`. It is best practice in object-oriented design to keep implementations of methods and fields that objects have in common in a mutual ancestor class.

Since `FamilyMember` is the superclass of both `Guardian` and `Sibling`, what instance fields and methods should it contain?

From now on you will never instantiate `FamilyMember`, and you will declare it `abstract` by modifying its class declaration to include the keyword `abstract` between `public` and `class`. This will create an error in `FamilyMemberFragment`. However, it no longer makes sense to have `FamilyMemberFragments`, now that you are not instantiating `FamilyMember`; `abstract` classes can never be instantiated.

20 Within your project, use the **Refactor** > **Rename** feature of Android Studio to replace `FamilyMemberFragment` with a new `GuardianFragment`. Refer to *Project 1.2.4P Create An Android App* if necessary.

21 Modify your `Guardian` class so that it extends `FamilyMember`.

`Guardian` now inherits everything from `FamilyMember,` and therefore, your `Guardian` class no longer needs whatever is defined in `FamilyMember`.

22 Remove the instance variables from `Guardian` that are defined in `FamilyMember`.

23 With first and last names defined in the Model layer of `FamilyMember`, add an `occupation` field to the `Guardian` Model.

24 Add features to display all `Guardian` data (inherited and otherwise) in the View and Presenter layers of your app. This will also help you differentiate `Guardians` from `Profiles`.

NOTE

This will cause a temporary error in `GuardianFragment`.

25 To fix the error in `GuardianFragment`, create a `Guardian` in place of the now abstract `FamilyMember`. Using the **Refactor** tool to change variable names from `mFamilyMember` to `mGuardian`.

Part IV: Date-of-Birth Picker

In this part of the project, you will add a date picker to your app so that a user can select their date of birth.

The instructions below provide steps to quickly place the date picker within the `ProfileFragment`. There is a set of longer instructions in **Optional Extension: Create a Date Picker Dialog** that also show you how to create a separate dialog to contain the date picker. Consult with your instructor to determine which instructions you should follow.

26 This set of instructions situates a `DatePicker` inside `ProfileFragment`:

a. Within `ProfileFragment`, create an instance field of type `DatePicker`.

b. Give the `DatePicker` whichever values you prefer for `layout_width` and `layout_height`.

c. Give the `DatePicker` an `id` attribute and name it descriptively.

d. Give the `DatePicker` a value of `false` for the `calendarViewShown` attribute.

e. In `ProfileFragment`, initialize your `DatePicker` using `rootView's` `findViewById` method as you did for other UI elements.

f. Test your code and layout to ensure you are content with the results. ***Hint***: *If your* `Button` *is not showing up, it may just be rendering behind your calendar. You can change the ordering in your layout file, or place all the elements in that layout file within a* `ScrollView`.

g. Add a `Date` object to your `Profile` class and use the internet to help you determine how to transfer the data from your `DatePicker` to that `Date` object.

CHALLENGES

Add the following features to your app:

☐ Data validation when your submit buttons are pressed

☐ A way to view what high school the applicant attended

☐ A way to view what standardized tests the applicant has completed

☐ A way to view information about one of the applicant's extracurricular activities (like hosting Fragments!)

☐ A way to view whether or not the applicant has submitted a personal essay

☐ A way to navigate between all previously added features

CONCLUSION

1. Why should `Guardian` and `Sibling` inherit from `FamilyMember`?
2. Why doesn't it make sense to instantiate `FamilyMember`?

Optional Extension: Create a Date Picker Dialog

Follow these optional instructions to create a pop-up **dialog** that prompts the user to enter a birthdate in the College App. A dialog is a small window box that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.

1. Create a `Button` in `ProfileFragment` and *`fragment_profile.xml`*; this `Button` will display the selected date of birth for the applicant and open a Date of Birth dialog. Give it an appropriate text attribute value.
2. Link the Presenter and View layers of this new `Button` together.
3. Call the `setOnClickListener` method of your new `Button`, placing the following code within the body of `onClick`:

   ```
   FragmentManager fm = getFragmentManager();
   DatePickerFragment dialog = new DatePickerFragment();
   dialog.show(fm, "DialogDateOfBirth");
   ```

   `FragmentTransactions` are one way to get your `Fragments` into your app; `FragmentManagers` are the other. If you have time, you should use the official documentation to determine the differences.
4. In the previous step you instantiated `DatePickerFragment`, a class you have yet to define. Create this class now, extending `DialogFragment`.
5. Override `DialogFragment's onCreateDialog(Bundle)` method in `DatePickerFragment`.
6. Within `onCreateDialog`, add the following code:

   ```
   View v = LayoutInflater.from(getActivity())
           .inflate(R.layout.dialog_date_of_birth, null);

   return new AlertDialog.Builder(getActivity())
           .setView(v)
           .setTitle("Date of Birth")
           .setPositiveButton("Done", null)
           .create();
   ```

If you have time, it is worth your while to research the Builder Pattern as it applies to Java and Android™ development.

7. Create a View layer file for your date picker.
8. Give the `DatePicker` whichever values you prefer for `layout_width` and `layout_height`.
9. Set the value of the `id` of the `DatePicker` to `"dialog_date_of_birth"`.
10. Give the `DatePicker` a value of `false` for the `calendarViewShown` attribute.
11. Test your app to make sure that the `DatePickerFragment` appears in a dialog when you press the blank button.

Next you will create a way to transfer a `Date` that will be stored in your `Profile` to the `DatePickerFragment` for display in your dialog. Add a constant `DATE_ARGUMENT` of type `String` with value `"date of birth"` to your `DatePickerFragment`.

12. Add an instance field of type `DatePicker` identified by `mDatePicker` to `DatePickerFragment`.
13. Add the following method to `DatePickerFragment`:

```java
public static DatePickerFragment newInstance(Date date) {
 Bundle args = new Bundle();
 args.putSerializable(DATE_ARGUMENT, date);

 DatePickerFragment fragment = new DatePickerFragment();
 fragment.setArguments(args);
 return fragment;
}
```

    This method will provide a `DatePickerFragment` with some customizations to client classes.

14. Add a `Date` object for date of birth to your `Profile` class and getters and setters for that object. Initialize it in your constructor using the `Date` constructor of your choice.
15. Within `ProfileFragment`, replace the right-hand side of the initialization of dialog with a call to the static method `newInstance` that you just created, passing in the applicant's date of birth from the Model layer. This gets the data from the Model through `ProfileFragment` to the new dialog.
16. In `onCreateDialog` within `DatePickerFragment`, Create a new `Date` object and initialize it to the `Date` (you will need to cast) returned by calling `getSerializable(DATE_ARGUMENT)` on the result of a call to `getArguments()`.
17. Create a new `Calendar` object in the same method and initialize it to the value returned by calling the static method `getInstance` of the `Calendar` class.
18. Call that `Calendar's setTime` method, passing in your `Date` object.

19. Add the following lines between the initialization of your `View` object and the `return` statement in `onCreateDialog`:

```
mDatePicker = (DatePicker) v.findViewById(R.id.dialog_date_of_birth);
mDatePicker.init(calendar.get(Calendar.YEAR), calendar.get(Calendar.MONTH),
    calendar.get(Calendar.DAY_OF_MONTH), null);
```

20. Test your code. The date that displays initially in your `DatePicker` should now be the value set in `Profile` in Step 14.

Next you will create a way for the `Date` selected within your `DatePicker` to be transferred back to `ProfileFragment` for display there. To do this, you will need to create a connection between `ProfileFragment` and `DatePickerFragment`.

21. In `ProfileFragment`, just after you initialize dialog, but before you call its show method, add a line of code that calls dialog's `setTargetFragment` method, passing in `ProfileFragment.this` as the first argument and `REQUEST_DATE_OF_BIRTH` as the second.
22. In `ProfileFragment`, create the constant `REQUEST_DATE_OF_BIRTH` that you used in the last step. It should have type `int` and value `0`.
23. In `DatePickerFragment`, add a `String` constant identified by `EXTRA_DATE_OF_BIRTH` with value `"org.pltw.examples.collegeapp.dateofbirth"`.

    This value will be used to identify what data you are passing from `DatePickerFragment` back to `ProfileFragment` on its termination.
24. Within `DatePickerFragment`, create a new helper method with return type `void` identified by `sendResult`, with an `int` parameter, `resultCode`, and a `Date` parameter, `date`.
25. As a first action within this method, create a conditional to make sure that this `DatePickerFragment` has a non-null value for `getTargetFragment()`. In the case that it does not, return out of the method.
26. After that first null check, create a new `Intent` object and initialize it using an empty constructor. `Intents` are the way that `Activitys` or `Fragments` can start new `Activitys` or `Fragments` to achieve a desired result while simultaneously passing "extra" information to those new `Activitys` or `Fragments`. More on this in later lessons.
27. Call the `putExtra` method of your new `Intent` object with `EXTRA_DATE_OF_BIRTH` as its first argument and `date` as its second.
28. Call `onActivityResult` with arguments `getTargetRequestCode()`, `resultCode`, and your new `Intent` object on the result of a call to `getTargetFragment()`. This will pass the new Intent back to `ProfileFragment`, along with the `Date` information that you stored by calling the `putExtra` in the previous step.

29. In `DatePickerFragment`, replace the `null` argument in the call to `setPositiveButton` with the following:

```
new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        Date date = new GregorianCalendar(mDatePicker.
        getYear(),
            mDatePicker.getMonth(), mDatePicker.
            getDayOfMonth()).getTime();
        sendResult(Activity.RESULT_OK, date);
    }
}
```

30. Within `ProfileFragment`, override `onActivityResult(int, int, Intent)`.
31. As a first step in this method, check to see if `resultCode` is equivalent to `Activity.RESULT_OK`. If it is not, then return.
32. Check next to see if `requestCode` is the same as `REQUEST_DATE_OF_BIRTH`.
33. If it is, call the `getSerializableExtra` method on your `Intent` parameter with the argument `DatePickerFragment.EXTRA_DATE_OF_BIRTH`.
34. Use the data from the previous step to set the date of birth in your model layer.
35. Set the text of your date of birth button to show the date of birth in its new form.
36. Test your app to make sure that it is behaving correctly.

Challenge

1. The date, as it displays on your button, is showing in a user-unfriendly manner. Format the date in a more intuitive fashion.