ACTIVITY **3.3.2**

Polymorphic People

INTRODUCTION

Polymorphism is a powerful tool for managing related yet distinct sets of data. Recall from Unit 1 how you created a polymorphic list of animals and used a single `say()` method to access the different animal objects in the same way. In this activity, you will use polymorphism for a list of contacts composed of family members, friends, classmates, and workmates and add them as your companions for a trip.

Materials

- Computer with Android™ Studio
- Android™ tablet and USB cable, or a device emulator
- Free Backendless account per student
- A Google account and a project on the Google API Console

RESOURCES

Activity 3.3.2 Visual Aid
Resources available online

Procedure

Part I: Prepare the Project

1. Open your TripTracker app in Android Studio.

NOTE

If you were unable to complete *Activity 3.3.1 Trip Cost and Rating*, import *3.3.1TripTracker_Solution* as directed by your teacher. Recall that if you import the solution, you must update keys values in *strings.xml:*

- Change be_app_id and be_android_api_key to your Backendless App ID and Android API key values, respectively. You can retrieve these from your **Backendless Console** (**Manage** icon).
- Change the value for google_app_id to your Google Play service API key. You can retrieve it from the **Google API Console** (**Credentials**).

2. If necessary, modify *strings.xml* to configure your project with your Backendless and Google API app IDs. Refer to *Activity 3.3.1 Trip Cost and Rating*.
3. If you have not created a few contacts on your device, do so now. (In later steps and activities, you will parse string data, so do not put strange characters, especially colons ":" or parentheses "( )", in the names of your contacts.)
4. Get a copy of *3.3.2TripTracker_StarterCode* from your teacher. Copy or extract the files to a temporary location.

The following steps will add a companions list to your Trip Details screen and define new activity for selecting contacts.

5. Copy the following XML files to TripTracker's *res/layout* directory.
   - *activity_contact.xml*
   - *companion_item.xml*
   - *fragment_contact_list.xml*
   - *fragment_contact_list_item.xml*
6. Copy *border_ui.xml* to TripTracker's *res/drawable* directory.
7. Copy *menu_contacts_list.xml* to TripTracker's *res/menu* directory.
8. Add the following to *strings.xml.*

```
<!-- 3.3.2 companions -->
<string name="companions_label">Trip Companions</string>
<string name="no_contacts_found_msg">No Contacts Found</string>
<string name="new_companion_button_label">Add</string>
<string name="contact_deselect_type">Deselect</string>
<string name="contact_family">Family</string>
<string name="contact_friend">Friend</string>
<string name="contact_school">School</string>
<string name="contact_work">Work</string>
```

9. Add the following values to *fragment_trip.xml* after the relative layout for dates.

```
<!-- 3.3.2 companions -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:background="@drawable/border_ui"
    >
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/companions_label"
            style="?android:listSeparatorTextViewStyle" />
        <ImageButton
            android:src="@android:drawable/ic_menu_add"
            android:id="@+id/new_companion_button_label"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            style="?android:attr/borderlessButtonStyle"
            android:scaleX=".7" android:scaleY=".7" />
    </LinearLayout>
    <ListView
        android:id="@+id/companion_listVew"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="15dp" />
</LinearLayout>
```

10. Also in *fragment_trip.xml*, use the sub-steps below to make the entire screen scrollable by adding a new view, a `ScrollView`, around your layout.

a. Add lines 12–20 to begin the `ScrollView`, which must include a new `LinearLayout`:

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <LinearLayout
 3:     xmlns:android="http://schemas.android.com/apk/res/android"
 4:     android:layout_width="match_parent"
 5:     android:layout_height="match_parent"
 6:     android:paddingBottom="@dimen/activity_vertical_margin"
 7:     android:paddingLeft="@dimen/activity_horizontal_margin"
 8:     android:paddingRight="@dimen/activity_horizontal_margin"
 9:     android:paddingTop="@dimen/activity_vertical_margin"
10:     android:orientation="vertical"><!-- 3.3.2 -->
11:
12:     <!-- 3.3.2 -->
13:     <ScrollView
14:          android:layout_width="match_parent"
15:          android:layout_height="match_parent">
16:
17:         <LinearLayout
18:             android:layout_width="match_parent"
19:             android:layout_height="match_parent"
20:             android:orientation="vertical">
```

b. To end the new `LinearLayout` and `ScrollView`, add lines 1–3 at the end of your file:

```
1:        <!-- 3.3.2 -->
2:            </LinearLayout>
3:    </ScrollView>
4:
5: </LinearLayout>
```

11. Copy the following Java files to TripTracker's *`java`* directory.
    - *`ContactActivity.java`*
    - *`ContactFragment.java`*
12. Define a new `ContactActivity` in `AndroidManifest.xml`.
    - The label is defined as `companions_label` in *`strings.xml`*.
    - The parent activity is TripActivity.
13. Also in *`AndroidManifest.xml`*, give permission for this `application` to read contacts from the device.

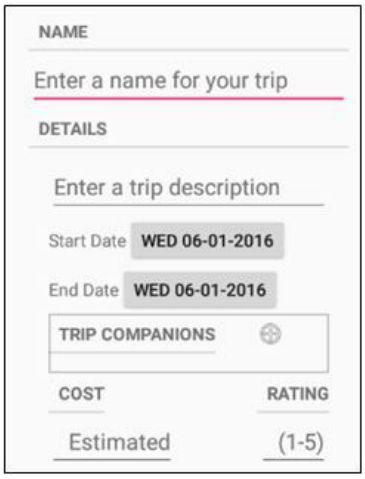
```
1: <uses-permission android:name="android.permission.READ_
   CONTACTS"/>
```

14. The new `ContactFragment` is responsible for adding companions to a trip. Review the flowcharts in the *3.3.2 Visual Aid* to get an overview of its functionality and relationship to `TripFragment`.

15. In your `ContactActivity` class, provide a way to ask permission to `READ_CONTACTS` like you did in `MapActivity` for location services. Be sure to define a new constant for tracking permission for contacts in place of `PERMISSION_LOCATION`.

16. Open *`Partial_TripFragment.java`* and follow the instructions in the comments for adding the methods and code fragments to your `TripFragment` class.

17. Test your app by creating a new trip and confirming that you have a companions list similar to the following.



PROJECT LEAD THE WAY
PLTW

Activity 3.3.2
Polymorphic People

Computer Science A

Recall that Polymorphism refers to the many different object types that can be processed in the same way.

In this activity, you will create subclasses from a parent Contact class. It will have the functionality that is common to all of the subclasses.

Each subclass inherits the instance fields and methods Contact. However they cannot directly access the them. Only the Contact class can directly access its own private variables. So, the Contact class provides accessors and mutators (getters and setters) for the subclasses to use. In this activity, they are: getDisplayName(), setDisplayName(...)

The subclasses can either:

Use the accessors and mutators provided in its parent class.

Or

Override the methods to implement their own, specific functionality.

Polymorphism Review

- Parent class

```
public class Contact {
     // common to all subclasses
     private String name;
     ...
}
```

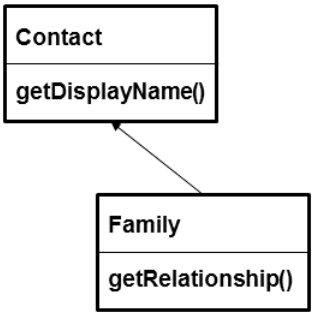
- Subclasses

```
public class Family extends Contact{
     // fields and methods
     // specific to a family member
     private String relationship;
     ...
}
```


Polymorphism Review – Super vs. Sub

- **Parent (super) class:** Cannot invoke subclass methods
- **Child (sub) classes:** Invoke parent methods via **`super`**

Writing subclass methods:
use **super** to access parent method
`super.getDisplayName();`



In polymorphic objects,
the super class does not
know about subclass methods:

```
Contact f = new Family();
f.getDisplayName(); // ok – f is a contact
f.getRelationship(); // error! A contact cannot
                     // see subclass methods
                     // instead cast it:
((Family) f).getRelationship(); // f actually references a Family
```

Part II: Create Parent and Subclasses

As you write new methods for this activity, be sure to provide Javadocs for them.

18. Review slides 2–3 of *3.3.2 Polymorphic People* presentation.

19. Create a new `Contact` class, and for now, leave the class empty. All of your companions will be one of the following subclasses of contact: Family, Friend, School, or Work. You will create these subclasses in the next step.

20. To create subclasses of `Contact` for `Family`, `Friend`, `School`, and `Work`, extend each class from the `Contact` class.

21. Create the following private instance fields with the specified data types:

    a. Family:

`relationship`	string
`birthday`	date (`java.util.Date`)

    b. Friend:

`knownfor`	double
`birthday`	date (`java.util.Date`)

    c. Work:

`title`	string
`companyName`	string

    d. School:

`classList`	array list of strings
`clubList`	array list of strings

22. Create all accessors and mutators that are specific to each of the subclasses you created and include Javadocs for all methods. (Start the Javadocs with `/**`).

    The mutators (setters) for `classList` and `clubList` in the `School` class will be different from the default mutators. They should add the club name (or class name), sent as a parameter, to the `School's ArrayList` of clubs (or classes). Reminder: You cannot use `class` as the name of an array list, since it is a keyword in Java. Instead use `classes`, `classList`, etc.

23. Return to *`Contact.java`* and create `String` instance variables for `name` and for `type`.

    Why do you suppose you are asked to create these in `Contact` and not in the subclasses?

24. Create the mutator and accessor methods (and Javadocs) for `name` and `type`.

25. Create a constructor for `Contact` that assigns `name` from a parameter.

Default Constructor – Review

- The **default constructor**: the constructor of the parent class when a constructor is not provided.
- The **Contact** class starts without a constructor, invoking the default constructor.
- When you write a constructor for **Contact**, you must also write constructors for all subclasses (thereby replacing the parent constructor).
- How to invoke them both in the subclass? Use **super()** in the subclass constructor.

Computer Science A

A feature of super and subclasses is the "default constructor". If you do not write a constructor for a class in Java, the constructor of its parent class will be invoked implicitly as the default constructor.

The Object class is the parent/super class of all objects in Java and all objects in Java inherit from it. Without an explicit constructor in a class (that is not itself a subclass), you are invoking the implicit constructor of its super class, in this case, Object.

Since you begin this activity without a constructor for the Contact class, the default constructor will be invoked when you construct a subclass of Family or Friend, for example. The subclass looks for a constructor in its parent class, and not finding one, uses the Object constructor to construct itself.

Later, when you do provide a constructor for the Contact class, it will have a string parameter displayName. All subclasses now must have a matching constructor as well, meaning a constructor with a string parameter. Why? You have effectively removed the default constructor that has no parameters. Remember, when attempting to construct a subclass without a constructor, you implicitly invoke the parent constructor. But you no longer have a matching constructor in Contact; you only have a constructor with a parameter. Without the default constructor in the parent class, and no explicit constructor in the subclass, the subclass will not know how to instantiate itself. Therefore, you must provide a matching constructor in the subclasses.

Often you want the functionality of both the parent class and subclass constructors: issue super (...) as the first line of the subclass' constructor. The parent constructor will be invoked followed by whatever functionality you provide in the subclass constructor.

26 Open one of your subclasses of `Contact` and note that creating a constructor in `Contact` introduced an error.

What does the error report and why?

27 Fix these errors by creating a constructor in each subclass that will invoke the `Contact` constructor using the `super` construct.

Explain why this fixed the error.

28. Also in each subclass constructor, invoke the `Contact setType` method to set the appropriate type of contact (such as “Family”, “Friend”, etc.) Check with your teacher to be sure your constructors are correct and then create Javadocs for them.

All subclasses will use `toString()` to display companion information for the Trip, specifically the `name` and `type` of the contact.

29. Rather than duplicate code in each `toString()` method of each subclass, create a `toString()` method in `Contact` and have it return `name + ": " + type`. Be sure to use the `@Override` nomenclature to verify that you are overriding correctly.

30. Explain how each subclass will use the `Contact toString()` method, and how values of `type` will be different.

Part III: Many Subclasses, One List

Now that you have defined a variety of contact subclasses, you will store them all in one array list. Contacts for a trip will be selected from a new screen called “Trip Companions”.

31. First, in *`ContactFragment.java`*, review the `getContacts` method and how a list called `mAllContacts` is populated from the contacts on your device.
    a. Review how `mAllContacts` is referenced in the `ContactsAdapter`.
    b. Review how contact types are added to the `mSelectedCompanionType` array in `onCheckedChanged`.
    c. Finally, observe `onCreateView` and find the four variables that store the string representation of the four contact subclasses.

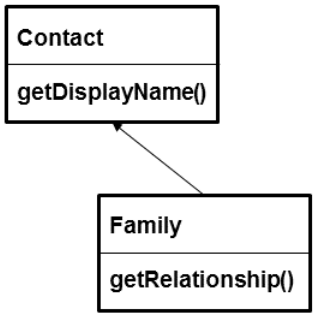
32. In `onOptionsItemSelected`, you have some TODO items. Read through this entire step and plan your algorithm before you start to write any code.
    a. Create a new array list of type `Contact`.
    b. Use the array `mSelectedCompanionType` to determine which contacts have been selected.
    c. Add the appropriate type of contact (Family, Friend, School, or Work) to the array list.
    d. Since an `ArrayList` of `Contact` objects will be put on the intent, make your `Contact` class implement `Serializable`. This will allow the intent to order the data in the array list properly.
    e. Put the polymorphic array list onto the intent. This will require a new constant in `IntentData`.

An important part of the algorithm is knowing that a variable reference can be `null`; each element of `mSelectedCompanionType` should be tested to make sure it is not null. Each non-null contact added to your array list should be an instance of `Family`, `Friend`, `School`, or `Work`.

Polymorphism Review – Super vs. Sub

- **Parent (super) class:** Cannot invoke subclass methods
- **Child (sub) classes:** Invoke parent methods via `super`

Writing subclass methods:
use **super** to access parent method
`super.getDisplayName();`



In polymorphic objects,
the super class does not
know about subclass methods:

```
Contact f = new Family();
f.getDisplayName(); // ok – f is a contact
f.getRelationship(); // error! A contact cannot
                     // see subclass methods
                     // instead cast it:
((Family) f).getRelationship(); // f actually references a Family
```


Default Constructor – Review

- The **default constructor**: the constructor of the parent class when a constructor is not provided.
- The **Contact** class starts without a constructor, invoking the default constructor.
- When you write a constructor for **Contact**, you must also write constructors for all subclasses (thereby replacing the parent constructor).
- How to invoke them both in the subclass? Use **super()** in the subclass constructor.

Dynamic/Late Binding - Review

```
ArrayList<Contact> a = new ArrayList<>();
a.add(new Family());
a.add(new Friend());
a.add(new School());
a.add(new Work());
...
a.get(i).toString();
```

- Which object's `toString` method gets invoked?
- With **dynamic binding**, the compiler determines which one.

Null Value

- Observe the object references that do not refer to anything

```
String s;
Animal a;
ArrayList<Animal> myList;
```

- What does each object reference contain?

  The **null value**
- It can be tested: `if (s != null) {...}`
- Otherwise, de-referencing `s`,
  as in `s.equals(someOtherString)`
  will result in a run-time `NullPointerException` error
- Compare to the **empty string** which is not null:

```
String s = "";
```

33 Test your progress by selecting a few of your contacts from the Trip Companions screen: You will *not* see any companions on the Trip Details screen yet, so use a log statement to verify that you are putting the correct companions on the intent.

34 Could you use an enhanced for loop to iterate over `mAllContacts`? Why or why not?

In terms of the app's functionality, what does a **null value** in `mSelectedContactType` indicate?

35 (Optional) You can improve the way contact types are managed. Instead of using string variables, create constants to represent contact types. Determine where these constants should be placed and their values.

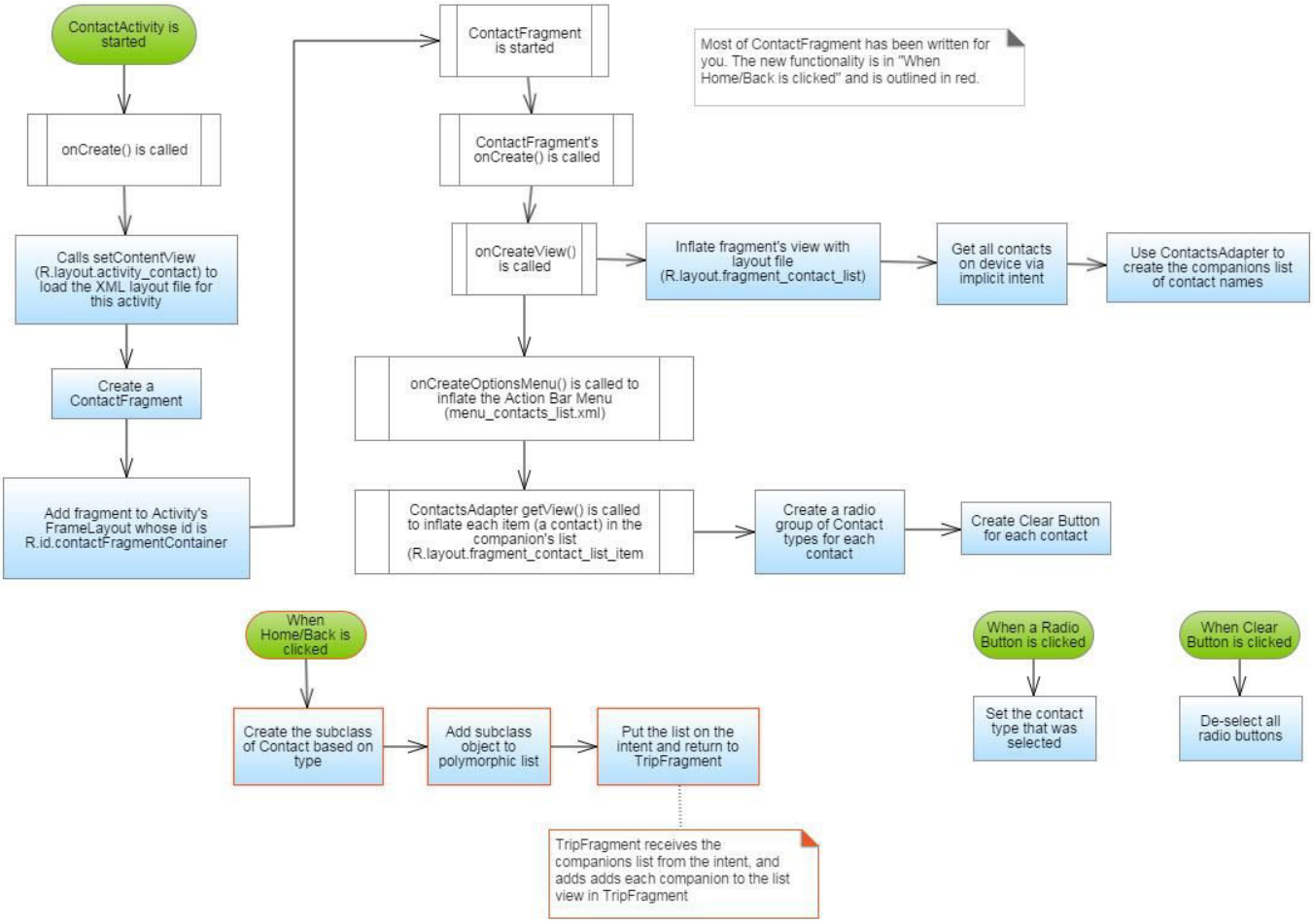
null value
The value assigned to an object reference when it has been created but not given an initial value.

CONCLUSION

1. Explain, in terms of dynamic binding, how an array list of type `Contact` can contain a Family object, Friend object, School object, and Work object. How is this a tool for abstraction?

Activity 3.3.2 Visual Aid

ContactFragment



TripFragment