Multiple Activities

Dr. Abraham (Abey) Campbell

Objectives

- Build an app involving several activities
- Manage the life cycle of an activity
- Use a RelativeLayout and a TableLayout
- Handle persistent data
- Animate the transition between screens

Mortgage App

- More Model-View-Controller
- Two Screens ➔ Two Activities
- RelativeLayout, TableLayout
- Go back and forth between the two Activities
- Pass data between Activities
- Handle persistent data
- Animated transition between screens

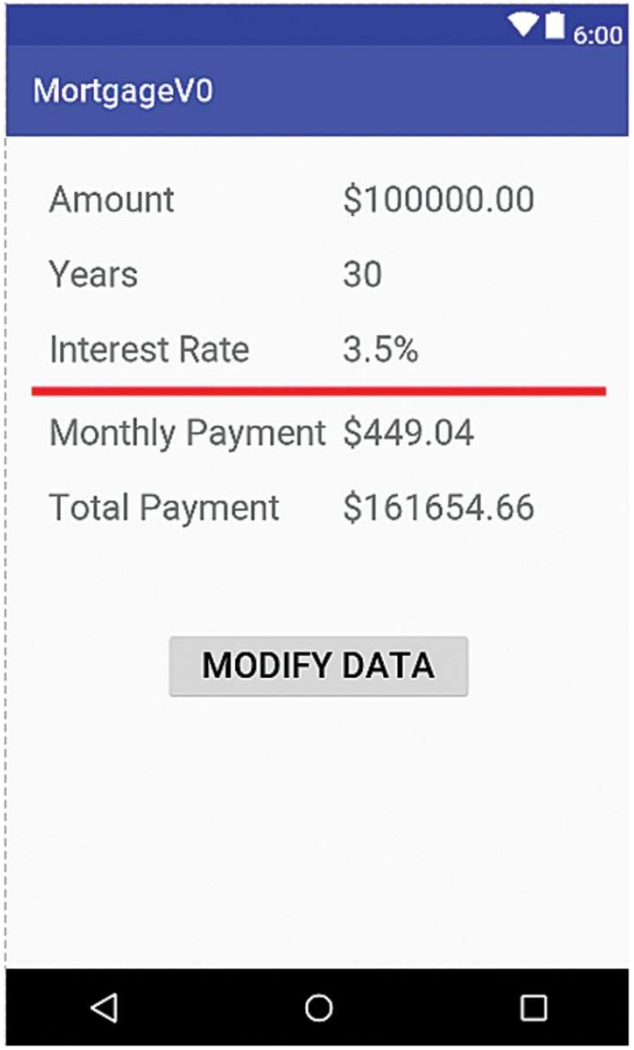
Mortgage—the Model

- The Mortgage class encapsulates a mortgage (Example 4.1).
- Amount, interest rate, number of years
- Methods to calculate the monthly payment, the total payments

Two Screens

- Screen 1: summary, monthly payment, total payment
- Screen 2: user input for amount, interest rate, number of years
- When going from screen 2 to screen 1, we update the data in screen 1.

Screen 1


6:00
MortgageV0
Amount
$100000.00
Years
30
Interest Rate
3.5%
Monthly Payment
$449.04
Total Payment
$161654.66
MODIFY DATA

TableLayout (1 of 8)

- We use a TableLayout for screen 1.
- TableLayout is a subclass of LinearLayout, itself a subclass of ViewGroup.
- A TableLayout arranges its children in rows and columns.

TableLayout (2 of 8)

- 6 rows, 2 columns
- First 5 rows: two TextViews: label and data
- Last row: button (to go to second activity/screen)

TableLayout (3 of 8)

- Typically uses a TableRow to define a row
- Inside a TableRow, we place two TextViews for the first 5 rows.
- We separate the first 3 rows (user input data) from the next 2 rows (calculated data) with a red line.

TableLayout (4 of 8)

```
<TableRow
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >
  <TextView
    android:text="@string/label_amount"
    android:padding="10dip" />
    <TextView
    android:id="@+id/amount"
    android:text="@string/amount" />
</TableRow>
```

TableLayout (5 of 8)

- We use strings defined in strings.xml as needed.
- We give ids to elements as needed.

TableLayout (6 of 8)

```
<!-- red line -->
<View
   android:layout_height="5dip"
   android:background="#FF0000" />
```

TableLayout (7 of 8)

- The last row is just one Button.
- We center it within the row.
- When the user clicks on the button, the modifyData method executes.

TableLayout (8 of 8)

```
<TableRow
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:paddingTop="50dip" >
    <Button
      android:text="@string/modify_data"
      android:onClick="modifyData" />
</TableRow>
```

strings.xml

- In Version 0, we hard code values in the strings.xml file.
- We do not use the Model. We fill the TextViews with those hard coded values.

styles.xml

- We edit styles.xml, defining a text size.
  <item name = "android:textSize">42sp
  </item>

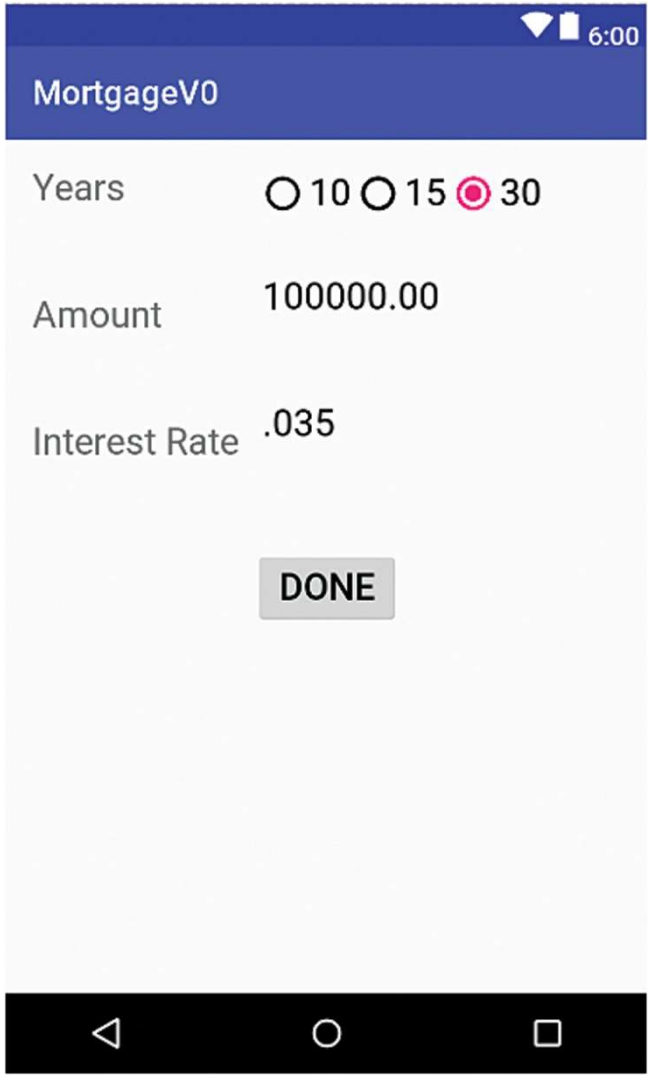
Screen 2 (1 of 3)

- Screen 2 is for user input.
- When the user clicks on the button of screen 1, we go to screen 2.
- We use a RelativeLayout.

Screen 2 (2 of 3)

- Number of years ➔ Three radio buttons (10, 15, 30 years)
- Mortgage amount ➔ EditText
- Interest rate ➔ EditText

Screen 2 (3 of 3)



Radio Buttons

- RadioButton class/XML element
- Each RadioButton element is placed inside a RadioGroup element ➔ they are mutually exclusive.

RadioGroup

```
<RadioGroup
   android:layout_toRightOf="@+id/label_years"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_alignLeft="@+id/data_rate"
   android:orientation="horizontal">
```

RadioButton

```
<RadioButton
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:id="@+id/ten"
   android:text="@string/ten" />
```

strings.xml

- We add strings to strings.xml for the elements in the second screen

...

<string name="thirty">30

</string>

<string name="amountDecimal">100000.00

</string>

...

Testing Screen 2

- We can look at screen 2 in the preview

OR

- We can change the View resource in MainActivity and run the app

```
// setContentView( R.layout.activity_main );
setContentView( R.layout.activity_data );
```

Connecting the Two Activities (1 of 2)

- In Version 1, we add code to MainActivity to go to the second activity and also come back to the first activity.
- We create a class for the second activity, DataActivity.
- We edit AndroidManifest.xml to include a second activity element.

Connecting the Two Activities (2 of 2)

- Activities are organized on a stack.
- What the user sees is the View of the Activity at the top of the stack.
- We go to another activity ➔ that activity is placed at the top of the stack.
- We finish an activity ➔ that activity is popped off the stack.

Activity Class Methods

public void startActivity(Intent intent)
Launches a new activity using intent as the Intent to start it.

public void finish( )
Closes this activity and pops it off the stack; the screen for the prior activity is shown.

Intent

Intent Constructor
Intent(Context context, Class<?> cls) Constructs an Intent that is intended to execute a class modeled by type ? (probably some Activity class) that is in the same application package as context

From Screen 1 to Screen 2 (1 of 2)

- Inside modifyData of MainActivity
- Create an Intent to go to a DataActivity
- Execute that Intent and start that DataActivity

From Screen 1 to Screen 2 (2 of 2)

```
public void modifyData( View v ) {
  Intent myIntent = new Intent( this,
            DataActivity.class );
  this.startActivity( myIntent );
}
```

Back to Screen 1 From Screen 2

(1 of 2)

- Finish current activity (this DataActivity)
- ➔ That pops it off the stack.
- ➔ The View for previous activity (MainActivity) the shows

Back to Screen 1 From Screen 2

(2 of 2)

- Inside DataActivity.

```
public void goBack( View v ) {
this.finish( );
}
```

AndroidManifest.xml (1 of 2)

- Add an activity element for the second activity (DataActivity)
- Place it inside the application element (like the first activity element)

AndroidManifest.xml (2 of 2)

```
<application …>
    <activity android:name=".MainActivity"
        …
    </activity>
    <activity
        android:name=".DataActivity"
        android:label="@string/app_name" >
    </activity>
```

Run the App

- Now we can go back and forth between the two screens.
- From screen 1 to screen 2, we push a DataActivity onto the activity stack.
- From screen 2 to screen 1, we pop it off the stack.

Life Cycle of an Activity

- An activity has a life.
- Some of its methods are called automatically (onCreate among others).
- ... When the activity goes in the background, comes back to the foreground, is finished ...

Life Cycle Methods of Activity (1 of 4)

- onCreate, onStart, onResume, onPause, onStop, onRestart, onDestroy are the life cycle methods of the Activity class.

Life Cycle Methods of Activity (2 of 4)

- onCreate: called when the activity is first created
- onStart: called after onCreate, when the activity becomes visible
- onResume: called after onResume, when the user starts interacting with the activity

Life Cycle Methods of Activity (3 of 4)

- onPause: called when Android starts or resumes another activity
- onStop: called when the activity becomes invisible to the user
- onRestart: called when the activity is about to restart
- onDestroy: called when the activity has ended

Life Cycle Methods of Activity (4 of 4)

- We can place Log statements inside each of these methods inside both Activity classes and check the Logcat output.
- We use a device, go back and forth between the two activities, let the current activity go in the background, then hit the Power button and swipe the screen to bring it to the foreground ... (see Figure 4.4).

Sharing Data between Activities (1 of 8)

- In Version 2, when we come back from screen 2 to screen 1, we update screen 1.
- We do the same when we go from screen 1 to screen 2.
- We pass data from screen 2 to screen 1 but also from screen 1 to screen 2.
- There are many ways of doing that.

Sharing Data between Activities (2 of 8)

- Use the putExtra methods of the Intent class.
- We can only pass primitive data types or Strings.
- ➔ not very convenient here because we want to pass the whole Mortgage object.

Sharing Data between Activities (3 of 8)

- An easy way to pass (or more exactly share) data between the two activities is to declare the Mortgage instance variable public and static in the MainActivity class.

  public static Mortgage mortgage;

- ➔ we can access it inside the DataActivity class using MainActivity.mortgage.

Sharing Data between Activities (4 of 8)

- We could also rewrite the Mortgage class as a singleton class.
- ➔ in this way, only one object of that class can be instantiated.
- ➔ it is automatically shared among the various activities.

Sharing Data between Activities (5 of 8)

- Use a file or a sqllite database to store the data.
- Read and write as needed.
- ➔ Either way, that is an overkill for this app.

Sharing Data between Activities (6 of 8)

- For this app, the simplest way to share data among activities is to make the mortgage instance variable public and static.
- Note: we will look at the putExtra method later in the book.

Sharing Data between Activities (7 of 8)

- In MainActivity, we add the updateView method to update the various TextViews based on data in the Model (mortgage instance variable).
- Call updateView inside onStart

Sharing Data between Activities (8 of 8)

- In DataActivity, add an updateView and an updateMortgageObject methods to:
  - Update the various GUI elements
  - Update the Model data after user input

Updating the GUI in Screen 1 (1 of 4)

- Use the findViewById method to retrieve a TextView.
- Update the text inside the TextView with data retrieved from the Model.

Updating the GUI in Screen 1 (2 of 4)

```
public void updateView( ) {
  // retrieve all the TextViews


  // update all the TextViews
  // based on the data inside the Model
}
```

Updating the GUI in Screen 1 (3 of 4)

```
public void updateView( ) {
  TextView amountTV =
       ( TextView ) findViewById( R.id.amount );
   amountTV.setText( mortgage.getFormattedAmount( ) );
   TextView yearsTV =
       ( TextView ) findViewById( R.id.years );
   yearsTV.setText( "" + mortgage.getYears( ) );
   …
}
```

Updating the GUI in Screen 1 (4 of 4)

- Call updateView every time the View shows.

```
public void onStart( ) {
super.onStart( );
updateView( );
}
```

Updating the GUI in Screen 2 (1 of 8)

- Same as for screen 1
- Use the findViewById method to retrieve a TextView or a RadioButton
- Update the text inside the TextView with data retrieved from the Model
- Check the radio button depending on state of Model

Updating the GUI in Screen 2 (2 of 8)

```
public void updateView( ) {
   // Get a reference to our Mortgage object
   Mortgage mortgage =MainActivity.mortgage;

   // update the View
}
```

Updating the GUI in Screen 2 (3 of 8)

```
// update the radio buttons
if( mortgage.getYears( ) == 10 ) {
    RadioButton rb10 =
        ( RadioButton ) findViewById( R.id.ten );
    rb10.setChecked( true );
} else if( mortgage.getYears( ) == 15 ) {
    RadioButton rb15 =
        ( RadioButton ) findViewById( R.id.fifteen );
    rb15.setChecked( true );
} // else do nothing (default is 30)
```

Updating the GUI in Screen 2 (4 of 8)

```
// update the EditTexts
EditText amountET =
   ( EditText ) findViewById( R.id.data_amount );
amountET.setText( "" + mortgage.getAmount( ) );
EditText rateET =
   ( EditText ) findViewById( R.id.data_rate );
rateET.setText( "" + mortgage.getRate( ) );
```

Updating the GUI in Screen 2 (5 of 8)

- Access the Mortgage object using MainActivity.mortgage
- Access the GUI elements using the findViewById method
- Retrieve user input
- Convert user input (a String) to a float as necessary
- Use the mutators of the Mortgage class to change the data inside the Mortgage object

Updating the GUI in Screen 2 (6 of 8)

```
public void updateMortgageObject( ) {
  Mortgage mortgage = MainActivity.mortgage;
  // update years in Model
  RadioButton rb10 = ( RadioButton ) findViewById( R.id.ten );
  RadioButton rb15 = ( RadioButton ) findViewById( R.id.fifteen );
  int years = 30;
  if( rb10.isChecked( ) )
       years = 10;
  else if( rb15.isChecked( ) )
       years = 15;
  mortgage.setYears( years );
  …
}
```

Updating the GUI in Screen 2 (7 of 8)

```
// update data amount in Model
EditText amountET = ( EditText ) findViewById(
R.id.data_amount );
String amountString = amountET.getText( ).toString( );
...
try {
    float amount = Float.parseFloat( amountString );
    mortgage.setAmount( amount );
    …
} catch( NumberFormatException nfe ) {
    mortgage.setAmount( 100000.0f );
    …
}
```

Updating the GUI in Screen 2 (8 of 8)

- We update the state of the Model before we go back to activity 1.

```
public void goBack( View v ) {
updateMortgageObject( );
this.finish( );
}
```

Transitions/Animations

- In Version 3, we enhance the user experience by setting up transitions between screens.
- Fade, scale, move left to right …
- Frame by frame or tween (automatically defined between a start frame and an end frame).

Animations

XML	Class	Description
set	AnimationSet	A set of several animations
alpha	AlphaAnimation	Fade in or out
Rotate	RotateAnimation	Rotate around a fixed point
scale	ScaleAnimation	Scale from a fixed point
translate	TranslateAnimation	Sliding animation

Transitions/Animations (1 of 2)

- We can use either XML or code to define animations.
- We use XML.

Transitions/Animations (2 of 2)

- Create a directory (anim) inside the res directory.
- Define animations in XML (one animation per XML file).
- Each XML element has attributes to define the animation.

slide_from_left.xml (1 of 2)

- Translation: main parameters are:
  - From position, to position, duration.

```
<translate
        android:fromXDelta="-100%p"
        android:toXDelta="0"
        android:duration="4000" />
```

slide_from_left.xml (2 of 2)

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android=
     "http://schemas.android.com/apk/res/android"
   android:interpolator=
     "@android:anim/accelerate_interpolator" >
   <translate
     android:fromXDelta="-100%p"
     android:toXDelta="0"
     android:duration="4000" />
</set>
```

fade_in_and_scale.xml (1 of 2)

- Can define several animations that will run concurrently
- We place them inside a set element.
- First, we define a fade animation.
- We follow it with a scale animation.

fade_in_and_scale.xml (2 of 2)

- Translation: main parameters
- From position, to position, duration

```
<set … >
        <alpha
         android:fromAlpha="0.0"
        android:toAlpha="1.0"
        android:duration="3000" />
        <scale
android:fromXScale="0.0"
 android:toXScale="1.0"
 … />
</set>
```

Specifying a Transition

- Now that we have defined animations, we can use them in order to transition from screen 1 to screen 2 and back from screen 2 to screen 1.

overridePendingTransition (1 of 2)

- Two animation resources are specified.
- One to enter the new activity and one to exit the current activity.

overridePendingTransition (2 of 2)

- We use this method from the Activity class to specify a transition from one activity to another.

void overridePendingTransition (int enterAnimResource, int exitAnimResource)

Inside MainActivity

```
public void modifyData( View v ) {
    Intent myIntent = new Intent( this,
                DataActivity.class );
    this.startActivity( myIntent );
    overridePendingTransition(
                R.anim.slide_from_left, 0 );
    /* 0 → no animation is used to transition
    from screen 1 */
}
```

Inside DataActivity

```
public void goBack( View v ) {
    updateMortgageObject( );
    this.finish( );
    overridePendingTransition(
            R.anim.fade_in_and_scale, 0 );
}
```

Managing Persistent Data (1 of 5)

- Next time we use the app, we want the app to remember the last mortgage data and use that data on the first and second screens.
- ➔ that data is persistent.
- In Version 4, we make the data persistent.

Managing Persistent Data (2 of 5)

- When the user uses the app for the first time, we show the default values for the three mortgage parameters: the mortgage amount, the interest rate, and the number of years.
- But when the user uses the app again, we want to show the values that were used the last time the user used the app.

Managing Persistent Data (3 of 5)

- In order to implement that functionality, we write to a file on the device the mortgage parameters every time they are changed.
- When we start the app the first time, the file does not exist and we use the default parameters for the mortgage. When we run the app afterwards, we read the mortgage parameters from the file.

Managing Persistent Data (4 of 5)

- Although we could use the openFileOutput and openFileInput methods of the ContextWrapper class to open a file for writing and reading, it is easier to use the user preferences system in order to store and retrieve persistent data.

Managing Persistent Data (5 of 5)

- Preferences for an app are organized as a set of key/value pairs, like a hashtable.
- In this app, since we have three values for a mortgage, we have three key/value pairs.

SharedPreferences

- The SharedPreferences interface includes the functionality to write to and read from the user preferences.
- Its static inner interface, Editor, enables us to store user preferences.

SharedPreferences.Editor

- We can use the putDataType methods of SharedPreferences.Editor to store data on the device. They have this general method header:

public SharedPreferences.Editor putDataType(String key, DataType value)

Method	Description
SharedPreferences.Editor putInt(String key, int value)	Associates value with key in this SharedPreferences.Editor. These key/value pairs should be committed by calling either the commit or apply method. Returns this SharedPreferences.Editor so that method calls can be chained.
SharedPreferences.Editor putFloat(String key, float value)	Associates value with key in this SharedPreferences.Editor. These key/value pairs should be committed by calling either the commit or apply method. Returns this SharedPreferences.Editor so that method calls can be chained.

SharedPreferences.Editor (1 of 2)

- Assuming we have a SharedPreferences.Editor reference named editor, in order to associate the value 10 with the key rating, we write:

```
// editor is a SharedPreferences.Editor
editor.putInt( "rating", 10 );
```

SharedPreferences.Editor (2 of 2)

- In order to actually write to the user preferences, we need to call the commit or apply method.

```
// editor is a SharedPreferences.Editor
editor.putInt( "rating", 10 );
editor.commit( );
```

SharedPreferences (1 of 5)

- To retrieve data previously written to the user preferences, we use the getDataType methods of the SharedPreferences interface.
- The getDataType methods have this general method header:

```
public DataType getDataType( String key,
        DataType defaultValue )
```

SharedPreferences (2 of 5)

Method	Description
int getInt(String key, int defaultValue)	Returns the int value associated with key in this SharedPreferences object; returns defaultValue if the key is not found.
float getFloat(String key, float defaultValue)	Returns the float value associated with key in this SharedPreferences object; returns defaultValue if the key is not found.

SharedPreferences (3 of 5)

- Assuming we have a SharedPreferences reference named pref, in order to retrieve the value that was previously associated with the key rating and written to the preferences, we write:

```
// pref is a SharedPreferences
// default value is 1
int storedRating = pref.getInt( "rating", 1 );
```

SharedPreferences (4 of 5)

- We can use the getDefaultSharedPreferences static method of the PreferenceManager class in order to get a SharedPreferences reference.

Method	Description
static SharedPreferences getDefaultSharedPreferences(Context context)	Returns the SharedPreferences for context.

SharedPreferences (5 of 5)

- Since the Activity class inherits from Context and our MainActivity and DataActivity classes inherit from Activity, we can pass the keyword *this* as the argument of this method.
- Thus, inside an Activity class, in order to get a SharedReferences inside our two classes, we can write:

```
SharedPreferences pref =
    ReferenceManager.getDefaultSharedPreferences( this );
```

SharedPreferences.Editor

- Once we have a SharedReferences reference, we can call the edit method of SharedPreferences to obtain a Sharedpreferences.Editor reference:

```
SharedPreferences pref = ReferenceManager
        .getDefaultSharedPreferences( this );
SharedPreferences.Editor editor = pref.edit( );
```

Impact on Our Classes (1 of 2)

- The View components of our app are still the same.
- Most of the changes take place in the Model, the Mortgage class.
- We also have small changes in the Controller classes, MainActivity and DataActivity.

Impact on Our Classes (2 of 2)

- We modify the Mortgage class so that it includes a method to write mortgage data to the user preferences system and a constructor to read data from it.
- In both the MainActivity and DataActivity classes, we use these methods to either load or write the mortgage parameters from and to the user preferences system.

Mortgage Class (1 of 2)

- Inside the constructor, we read the preferences and assign the corresponding values to the three instance variables: amount, rate, and years.
- If it is the first time the app is run, there are no preferences set; we assign default values to the three instance variables.

Mortgage Class (2 of 2)

- Since we need three keys (for three values that we store in the preferences), we define three constants for them:

```
private static final String PREFERENCE_AMOUNT
    = "amount";
private static final String PREFERENCE_YEARS
    = "years";
private static final String PREFERENCE_RATE
    = "rate";
```

Mortgage Constructor

```
public Mortgage( Context context ) {
    SharedPreferences pref =    PreferenceManager
                .getDefaultSharedPreferences( context );
    setAmount( pref.getFloat( PREFERENCE_AMOUNT,
                100000.0f ) );
      setYears( pref.getInt( PREFERENCE_YEARS, 30 ) );
      setRate ( pref.getFloat( PREFERENCE_RATE, 0.035f ) );
}
```

Mortgage Class (1 of 2)

- We include the method setPreferences: it writes to the preferences.
- We include a Context parameter so we can pass it to the getDefaultSharedPreferences method.

```
public void setPreferences( Context context )
```

Mortgage Class (2 of 2)

- When we call the setPreferences method from the DataActivity class using the Mortgage object reference mortgage, we will pass *this*.

```
mortgage.setPreferences( this );
```

setPreferences Method (1 of 3)

```
public void setPreferences( Context context ) {
    // Get a SharedPreferences.Editor
    // With it, write the data to the preferences
}
```

setPreferences Method (2 of 3)

- First we get a SharedPreferences.Editor reference.

```
SharedPreferences pref =
   PreferenceManager.getDefaultSharedPreferences( context );
SharedPreferences.Editor editor = pref.edit( );
```

setPreferences Method (3 of 3)

- Next, we write the data into the preferences.
- Amount, years, and rate are the three instance variables of the Mortgage class.

```
editor.putFloat( PREFERENCE_AMOUNT, amount );
editor.putInt( PREFERENCE_YEARS, years );
editor.putFloat( PREFERENCE_RATE, rate );
editor.commit( );
```

MainActivity—onCreate

- Inside the onCreate of MainActivity, we instantiate the Mortgage instance variable with this statement (an Activity "is a" ContextWrapper):

```
mortgage = new Mortgage( this );
```

DataActivity—updateMortgageObject

- Inside the updateMortgageObject method, after calling the mutators to update the Mortgage data, we call setPreferences so that we write the current Mortgage data to the preferences.

```
mortgage.setPreferences( this );
```

AndroidManifest.xml

- We need to specify in the manifest that this app writes to the device file system
- We add a uses-permission element inside the manifest element as follows:

<uses-permission android:name=
"android.permission.WRITE_EXTERNAL_STORAGE" />