Mobile Application Development

Model-View-Controller in Android

In this Lecture, you will learn:

➢ Creating new classes
➢ MVC Architecture in Android
➢ Updating Layers of MVC
➢ Adding Icon

In this lecture, we will update the application to add more questions and new buttons to view previous and next questions. Model View Controller Architecture will be introduced. Figure 1 shows next button added to application. Adding image and icon resources will also be discussed.

Figure 1: Adding a new button for next quiz

Adding new classes:

In MVC architecture, Model refers to the data. For the GeoQuiz app, the main data is the quiz and its correct answer. For this we need to define its Model. This data can be modeled as a class.

We are going to add a class named Question to the project. An instance of this class will encapsulate a single true-false question.

We will create an array of Question objects to accommodate more questions.

In the project tool window, right-click the application package and select New → Java Class. Name the class Question and click OK to create the new class. Since this class will hold question and correct answer, so this class serves as the model.

In Question.java, add two member variables and a constructor.

class (Question.java)

```
public class Question {
        private int mTextResId;
        private boolean mAnswerTrue;
        public Question(int textResId, Boolean answerTrue) {
                mTextResId = textResId;
                mAnswerTrue = answerTrue;
        }
}
```

Listing 1: Creating new question class

Question class holds the question text and the question answer (true or false). The mTextResId variable is int and not String because it will hold the resource ID (always an int) of a string resource for the question.

These variables need getter and setter methods which can be auto generated by Android Studio.

Generating getters and setters

Configure Android Studio to recognize the m prefix for member variables. Open Android Studio's preferences (File → Settings on Windows). Expand Editor and then expand Code Style. Select Java, then choose the Code Generation tab.

In the Naming table, select the Field row and add m as the name prefix for fields (Figure 2). Add s as the name prefix for static fields.

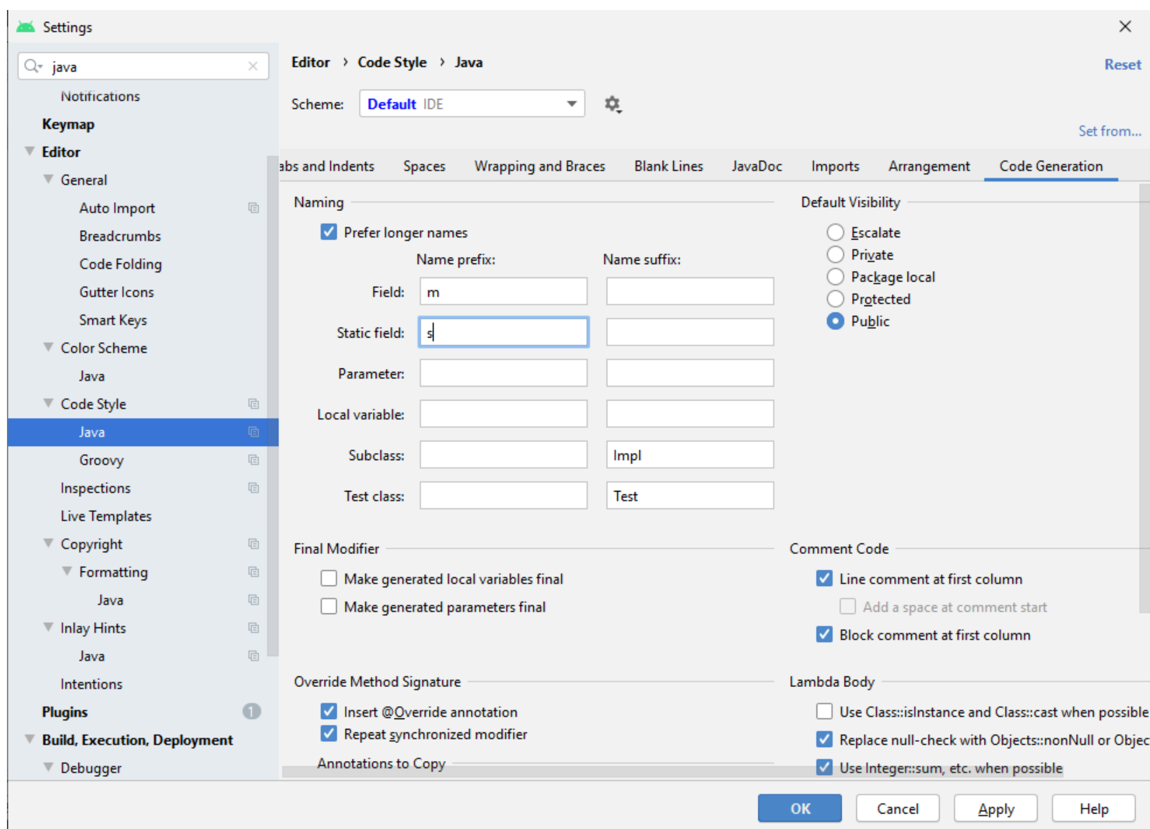

Figure 2: Setting Java code style preferences

Now generating a getter for mTextResId will create getTextResId() rather than getMTextResId() and isAnswerTrue() rather than isMAnswerTrue().

In Question.java, right-click after the constructor and select Generate... and then Getter and Setter. Select mTextResId and mAnswerTrue and click OK to create a getter and setter for each variable. This will generate the code as shown in listing 2.

```
public class Question {
        private int mTextResId;
        private boolean mAnswerTrue;
        ...
        public int getTextResId() {
                return mTextResId;
        }
        public void setTextResId(int textResId) {
                mTextResId = textResId;
        }
        public boolean isAnswerTrue() {
                return mAnswerTrue;
        }
        public void setAnswerTrue(boolean answerTrue) {
                mAnswerTrue = answerTrue;
        }
}
```

Listing 2: Auto generated getters and setters

The Question class is now complete. Now modify QuizActivity to work with Question. In QuizActivity create an array of Question objects. It will then interact with the TextView and the three Buttons to display questions and provide feedback. Figure 3 shows these relationships.

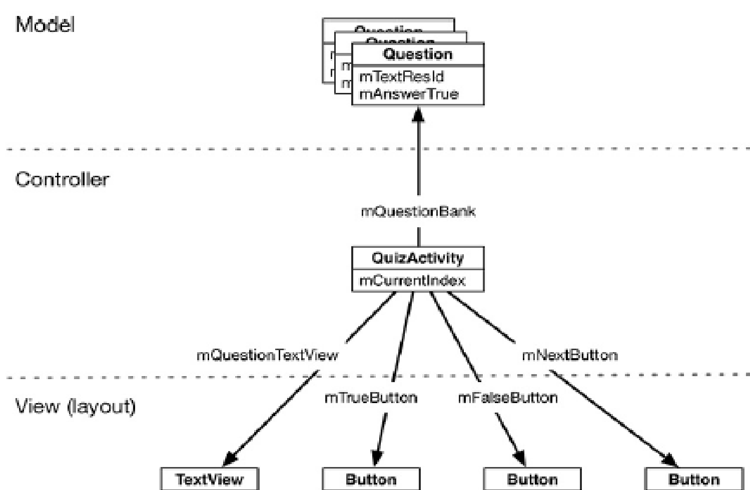

Figure 3: Object diagram showing MVC architecture of the app

Model-View-Controller and Android

In Figure 3 objects are separated as Model, Controller, and View. Android applications are designed around an architecture called Model-View-Controller, or MVC. In MVC, all objects in your application must be a model object, a view object, or a controller object.

A **model** object holds the application's data and "business logic." It models the things your app is concerned with, such as a user, a product, a photo on a server, or a true-false question. Model objects have no knowledge of the UI; their sole purpose is holding and managing data. GeoQuiz's model layer consists of the Question class.

View objects draw themselves on the screen and respond to user input, like touches. You can also create custom view classes. An application's view objects make up its view layer. GeoQuiz's view layer consists of the widgets that are inflated from activity_main.xml.

Controller objects bind the view and model objects. They contain "application logic." Controllers are designed to respond to events triggered by view objects and to manage the flow of data to and from model objects and the view layer. Controller in Android is typically a subclass of Activity, Fragment, or Service. GeoQuiz's controller layer, is the MainActivity at this point.

MVC Architecture Benefits

Separating classes into model, view, and controller layers helps you design and understand an application; you can think in terms of layers instead of individual classes.

You can see the benefits of keeping layers separate in a simple app like GeoQuiz. To update GeoQuiz's view layer to include a NEXT button, you do not need to remember a anything about the Question class.

MVC also makes classes easier to reuse. A class with restricted responsibilities is more. For instance, the Question model class, knows nothing about the widgets used to display a true-false question. This makes it easy to use Question throughout your app for different purposes. For example, if you wanted to display a list of all the questions at once, you could use the same object that you use here to display just one question at a time.

Updating the View Layer

Objects in the view layer are inflated from XML layout file. The activity_main.xml layout needs to be updated to add Next Button.

The changes you need to make to the view layer are:
Remove the android:text attribute from the TextView. You no longer want a hardcoded question. Give the TextView an android:id attribute to give it a resource ID so that you can set its text in the Activity's code. Add the new Button widget as a child of the root Layout.

```
...
<string name="question_australia">Canberra is the capital of Australia.
</string>
<string name="question_oceans">The Pacific Ocean is larger than
the Atlantic Ocean.</string>
<string name="question_mideast">The Suez Canal connects the Red Sea
and the Indian Ocean.</string>
<string name="question_africa">The source of the Nile River is in
Egypt.</string>
<string name="question_americas">The Amazon River is the longest river
in the Americas.</string>
<string name="question_asia">Lake Baikal is the world\'s oldest and
deepest freshwater lake.</string>
<string name="true_button">True</string>
<string name="false_button">False</string>
<string name="next_button">Next</string>
<string name="correct_toast">Correct!</string>
```

Listing 3 Updating strings (strings.xml)

Update the strings.xml file to add questions text as shown in listing 3.

Updating the Controller Layer

Open QuizActivity.java. Add variables for the TextView and the new Button. Also, create an array of Question objects and an index for the array.

```
public class QuizActivity extends AppCompatActivity {
private Button mTrueButton;
private Button mFalseButton;
private Button mNextButton;
private TextView mQuestionTextView;
private Question[] mQuestionBank = new Question[]
{
        new Question(R.string.question_australia, true),
        new Question(R.string.question_oceans, true),
        new Question(R.string.question_mideast, false),
        new Question(R.string.question_africa, false),
        new Question(R.string.question_americas, true),
        new Question(R.string.question_asia, true),
};
private int mCurrentIndex = 0;
...
```

Listing 4: Adding variables and Question array to controller

Here Question constructor is invoked several times to create an array of Question objects.

Use mQuestionBank, mCurrentIndex, and the accessor methods in Question to show questions on screen. First, get a reference for the TextView and set its text to the question at the current index.

```
public class QuizActivity extends AppCompatActivity {
        ...
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                mQuestionTextView = (TextView)
                findViewById(R.id.question_text_view);
```

```
        int question =
        mQuestionBank[mCurrentIndex].getTextResId();
        mQuestionTextView.setText(question);
        mTrueButton = (Button)
        findViewById(R.id.true_button);
        ...
    }
}
```

Listing 5: Wiring up the TextView in controller

Now wire up NEXT button to increment the index and update the TextView's text to show next question.

```
public class QuizActivity extends AppCompatActivity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
    ...
    mFalseButton.setOnClickListener(new View.OnClickListener() {
        ...
        }
    });
    mNextButton = (Button) findViewById(R.id.next_button);
    mNextButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            mCurrentIndex = (mCurrentIndex + 1) %
            mQuestionBank.length;
            int question =
            mQuestionBank[mCurrentIndex].getTextResId();
            mQuestionTextView.setText(question);
        }
    });
    }
}
```

Listing 6: Wiring up the new button (MainActivity.java)

Put this code into a private method as shown in Listing 7 and call that method in the mNextButton's listener and at the end of onCreate(Bundle) to initially set the text in the activity's view.

```
public class QuizActivity extends AppCompatActivity {
        ...
        @Override
        protected void onCreate(Bundle savedInstanceState) {
        ...
        mQuestionTextView = (TextView)
        findViewById(R.id.question_text_view);
        int question =
        mQuestionBank[mCurrentIndex].getTextResId();
        mQuestionTextView.setText(question);
        ...
        mNextButton.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                        mCurrentIndex = (mCurrentIndex + 1) %
                        mQuestionBank.length;
                        int question =
                        mQuestionBank[mCurrentIndex].getTextResId();
                        mQuestionTextView.setText(question);
                        updateQuestion();
                }
        });
                updateQuestion();
        }
        private void updateQuestion() {
                int question =
                mQuestionBank[mCurrentIndex].getTextResId();
                mQuestionTextView.setText(question);
        }
}
```

Listing 7 Encapsulating with updateQuestion()

Now it is time to turn to the answers. Create a new method that will accept a Boolean variable that identifies whether the user pressed TRUE or FALSE. It will check the user's answer against the answer in the current Question object and will make a Toast.
In MainActivity.java, add checkAnswer(boolean) shown in Listing 8.

```
public class QuizActivity extends AppCompatActivity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
    ...
    }
    private void updateQuestion() {
        int question = mQuestionBank[mCurrentIndex].getTextResId();
        mQuestionTextView.setText(question);
    }
    private void checkAnswer(boolean userPressedTrue)
    {
        boolean answerIsTrue =
        mQuestionBank[mCurrentIndex].isAnswerTrue();
        int messageResId = 0;
        if (userPressedTrue == answerIsTrue) {
            messageResId = R.string.correct_toast;
        } else {
            messageResId = R.string.incorrect_toast;
        }
        Toast.makeText(this, messageResId, Toast.LENGTH_SHORT)
        .show();
    }
}
```

Listing 8 Adding checkAnswer method

Now call this method from each of the buttons True and False to check the answer is shown in listing 9.

```
public class QuizActivity extends AppCompatActivity {
        ...
        @Override
        protected void onCreate(Bundle savedInstanceState) {
        ...
        mTrueButton = (Button) findViewById(R.id.true_button);
        mTrueButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                Toast.makeText(QuizActivity.this, R.string.correct_toast,
                Toast.LENGTH_SHORT).show();
                checkAnswer(true);
        }
        });
        mFalseButton = (Button) findViewById(R.id.false_button);
        mFalseButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                Toast.makeText(QuizActivity.this, R.string.incorrect_toast,
                Toast.LENGTH_SHORT).show();
                checkAnswer(false);
        }
        });
        ...
        }
        ...
}
```

Listing 9: calling checkAnswer method

Adding an Icon

GeoQuiz is now running, but the UI would be better with NEXT button having right-pointing arrow icon.

Though the icons are available online, the following github project provides a number of useful icons.
https://github.com/ksoichiro/Android-ColorfulIcons

In your project, within this directory are the drawable-hdpi, drawablemdpi, drawable-xhdpi, and drawable-xxhdpi directories.
The suffixes on these directory names refer to the screen pixel density of a device:

mdpi medium-density screens	(~160dpi)
hdpi high-density screens	(~240dpi)
xhdpi extra-high-density screens	(~320dpi)
xxhdpi extra-extra-high-density screens	(~480dpi)

Within each directory, you should have two image files – arrow_right.png and arrow_left.png, customized for the screen pixel density specified in the directory's name. Include all the image files in GeoQuiz.
When the app runs, the OS will choose the best image file for the specific device running the app.
Note that by duplicating the images increases the app size a bit.

If an app runs on a device that has a screen density not included in any of the application's screen density qualifiers, Android will automatically scale the available image to the appropriate size for the device. Thus it is not necessary to provide images for all of the pixel density buckets. To reduce the size of your application, you can only include a few of the higher resolution buckets and selectively optimize for lower resolutions when Android's automatic scaling provides an image with artifacts on those lower resolution devices.

Adding resources to a project

The next step is to add the image files to GeoQuiz's resources.
Open project tool window, displaying the Project view (select Project from the dropdown at the top of the project tools window). Expand the GeoQuiz/app/src/main/res.

Copy the png icons and paste them copied the corresponding directories within app/src/main/res. The filenames must be lowercase and have no spaces. You should now have four density qualified directories, each with an arrow_left.png and arrow_right.png file.

Switch back to the Android view, to see the newly added drawable files.

Any .png, .jpg, or .gif file you add to a res/drawable folder will be automatically assigned a resource ID. When the app runs, the OS will determine the appropriate image to display on that particular device.

Referencing resources in XML

Open activity_quiz.xml and add two attributes to the Button widget definition.

```
…
<Button
android:id="@+id/next_button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/next_button"
android:drawableRight="@drawable/arrow_right"
android:drawablePadding="4dp" />
…
```

Listing : Adding an icon to the NEXT button (activity_quiz.xml)

In an XML resource, you refer to another resource by its resource type and name. A reference to a string resource begins with @string/. A reference to a drawable resource begins with @drawable/.

The ImageButton

Use ImageButtons instead of regular Buttons to show just icons without text. ImageButton is a widget that inherits from ImageView. Button, on the other hand, inherits from TextView.