Mobile for the masses: Words and gestures with Overheard Word

Programmatically integrate third-party code into your Android UI

Andrew Glover
Author and developer
App47

Skill Level: Introductory

Date: 28 Jun 2013

You might feel like a kid in a candy store grabbing third-party code from GitHub or another repository, but there are still some tricks to integrating it with your Android UI. This month, Andrew Glover shows you how to take the Overheard Word demo app up a level with a JSON-based word engine and some prebaked, swipe-gesture functionality. As it turns out, Android easily accommodates third-party code, but you still have to do some careful logic if you want your app's UI to run it smoothly.

View more content in this series

If you've been reading and following along with the demos so far in this series, then you've established some basic Android development skills. In addition to setting up an Android development environment and writing your first Hello World app, you've learned how to replace button taps with swipe gestures and implement menus (or action bars) and icons in a custom mobile application. In this article, you continue along that trajectory by learning how to use third-party libraries to increase or enhance an app's functionality. First, we'll install a couple of open source libraries and read the files, then we'll programmatically integrate that new functionality with a demo app's UI.

As I did in previous articles, I'll be using my Overheard Word app for demonstration purposes. If you haven't already done so, you should clone the GitHub repository for Overheard Word so that you can follow along.

About this series

Mobile application distribution is exploding, and so is the market for mobile development skills. This new series introduces the mobile landscape to developers who are experienced with programming but new to mobility.

Start out coding native apps in Java code, then expand your toolkit to include JVM languages, scripting frameworks, HTML5/CSS/JavaScript, third-party tools, and more. Step by step, you'll master the skills you need to meet virtually any mobile development scenario.

Thingamajig: A pluggable word engine

Overheard Word is an English-language app that helps users learn new words and build vocabulary on the fly. In previous articles, we started with developing a basic app, then we added swipe gestures for easier navigation and icons for a shinier UI. So far so good, but this app isn't going very far without a certain missing ingredient: Overheard Word needs some words!

In order to make Overheard Word a truly *wordy* app, I've built a small word engine, Thingamajig, that encapsulates the notion of a word and its corresponding definitions. Thingamajig handles creating words and their definitions via JSON documents and is in no way dependent on Android. Like the Overheard Word app, my word engine is hosted on GitHub. You can clone the repository or download the source, then run `ant`.

You'll end up with an 8KB lightweight JAR file, which you can copy into your `libs` directory. If your Android project is set up right, your IDE will automatically recognize any file in the `libs` directory as a dependency.

For the time being, the code in my word engine is initialized via a JSON document instance. The JSON document could be a file residing on a file system on a device, a response to an HTTP request, or even a response to a database query — for now it really doesn't matter. What does matter is that you have a nifty library that allows you to work with `Word` objects. Each `Word` object contains a collection of `Definition` objects and a corresponding part of speech. While there's more to words and their definitions than this simple relationship, it will work for now. Later we can add add example sentences, synonyms, and antonyms.

Third-party libraries in Android

It's pretty easy to integrate a third-party library into Android projects; in fact, every Android startup project includes a special directory, `libs`, where you can place third-party JARs. During the Android build process, both normal JVM files and third-party JARs are converted to be compatible with Dalvik, the specialized Android VM.

Environmental constraint

While you could conceivably add any third-party library you like to your app, never forget the the limitations of a mobile environment. The device your app is running on isn't some beefy web server, after all! Users will thank you for conserving processing power and minimizing the download size of third-party add-ons.

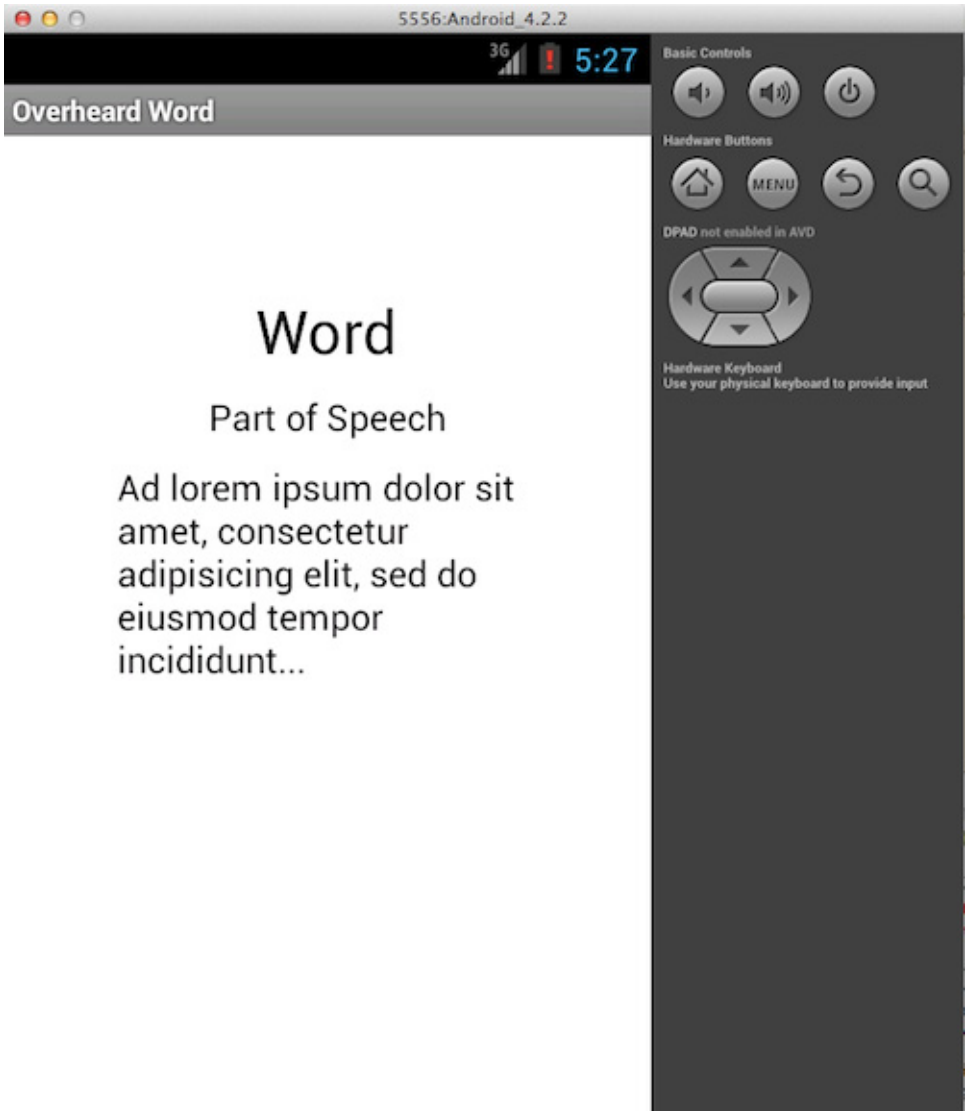
In addition to plugging the word engine library into my app, I'm going to add *another* third-party app called Gesticulate. Like with the word engine library, you can get

Gesticulate by cloning or downloading its source code from GitHub. Then run `ant` and you'll get a JAR file, which you can throw into your app's `libs` directory.

Updating the UI

Now we get into the heart of this article, which is about updating your UI to integrate all that third-party code you just snapped up for free. Fortunately, I planned ahead for this moment with my Overheard Word app. Back when I wrote the first iteration of the app, I defined a simple layout consisting of a word, its part of speech, and a definition, as shown in Figure 1:

Figure 1. Overheard Word's default view



The layout is defined with placeholder text, which I'm going to replace with actual words obtained from my pluggable word engine. So I'll initialize a series of words, grab one, and use its values to update the UI accordingly.

To update the UI, I have to get a handle to the individual view elements and provide them with values. For example, a displayed word, like *Pedestrian*, is defined as a `TextView` whose ID is `word_study_word`, as you can see in Listing 1:

Listing 1. TextView defined in a layout

```
<TextView
       android:id="@+id/word_study_word"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_gravity="center"
       android:layout_marginBottom="10dp"
       android:layout_marginTop="60dp"
       android:textColor="@color/black"
       android:textSize="30sp"
       android:text="Word"/>
```

If you look closely, I've set the text to "`Word`" in the XML; however, I can programmatically set that value by obtaining a reference to the `TextView` instance and calling `setText`.

Before I can proceed any further, I need to build a list of words. You might remember that my word engine can create `Word` instances via JSON documents, so all I need to do is set up my Android app to include and read these custom files.

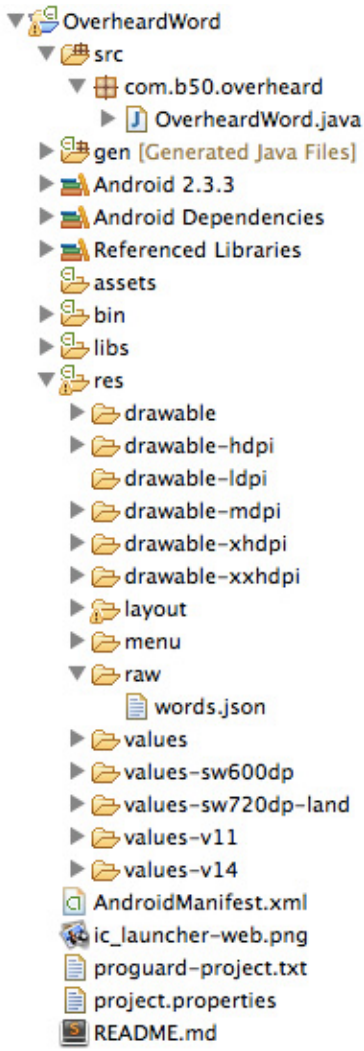
Working with files: The res and raw directories

An Android app, by default, has access to read the underlying file system of a device, but you only have access to the local file system underneath the app itself. Thus, you can include files in your app and reference them accordingly. (This means that you can read and write files *local to your app*; writing to the file system outside of your app, such as an SD Card, requires specialized permissions.) Because my word engine can take a JSON document to initialize a list of words, there is nothing to stop me from including a JSON document that my app will read at runtime.

Like the icon assets I presented in my previous article, you can store files in the `res` directory. If I find myself in need of custom files, I like to add them into a directory dubbed `raw`. Any file I drop in that directory can be referenced via the generated `R` file that I've already used a few times for Overheard Word. Basically, the Android platform takes assets from the `res` directory and builds a class called `R`, which then provides a handle to those assets. If the asset is a file, then the `R` file will provide a reference to open the file and obtain its contents.

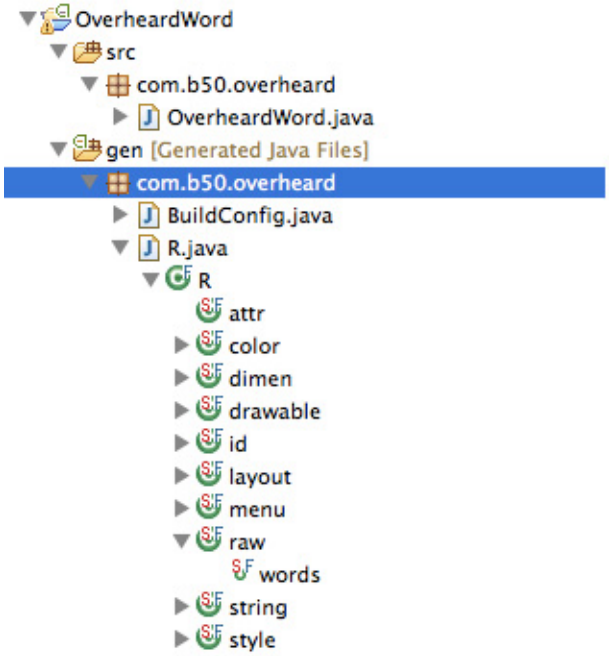
I create a `raw` directory within the `res` directory structure and place a JSON document in it as shown in Figure 2:

Figure 2. The raw directory with new words



Next, Eclipse rebuilds the project, and my `R` file gets a handy reference to the new file, as shown in Figure 3:

Figure 3. The updated R file



Once I have a handle to the file, I can open it, read it, and ultimately build a JSON document to serve as the basis for generating a list of words.

Building a list of words

When the app starts up, I'll fire off a series of steps that loads the raw JSON document and builds a list of words. I'll create a method dubbed `buildWordList` to handle this, as you can see in Listing 2:

Listing 2. Reading a file's contents in Android

```
private List<Word> buildWordList() {
  InputStream resource =
    getApplicationContext().getResources().openRawResource(R.raw.words);
  List<Word> words = new ArrayList<Word>();
  try {
    StringBuilder sb = new StringBuilder();
    BufferedReader br = new BufferedReader(new InputStreamReader(resource));
    String read = br.readLine();
    while (read != null) {
        sb.append(read);
        read = br.readLine();
    }
    JSONObject document = new JSONObject(sb.toString());
    JSONArray allWords = document.getJSONArray("words");
    for (int i = 0; i < allWords.length(); i++) {
        words.add(Word.manufacture(allWords.getJSONObject(i)));
    }
  } catch (Exception e) {
    Log.e(APP, "Exception in buildWordList: " + e.getLocalizedMessage());
  }
  return words;
}
```

You should notice a few things going on in the `buildWordList` method. First, note how the `InputStream` is created using Android platform calls that ultimately reference the

`words.json` file found in the `raw` directory. Nowhere do I use a `String` representing a path, which makes the code portable across a large swath of devices. Next, I'm using a simple JSON library *included* in the Android platform to convert the contents (as represented by a `String`) into a series of JSON documents. The static method `manufacture` on `Word` takes a single JSON document representing a word.

The JSON format for a word is shown in Listing 3:

Listing 3. JSON represents a word

```
{
  "spelling":"sagacious",
  "definitions": [
     {
      "part_of_speech":"adjective",
      "definition":"having or showing acute mental discernment
        and keen practical sense; shrewd"
     }
    ]
}
```

My `WordStudyEngine` class is the main facade for Thingamajig. It produces random words and functions off of a list of `Word` instances. So my next step is to initialize the engine with the newly built `Word List`, as shown in Listing 4:

Listing 4. Initializing the WordStudyEngine

```
List<Word> words = buildWordList();
WordStudyEngine engine = WordStudyEngine.getInstance(words);
```

Once I have an initialized engine instance, I can ask it for a word (which is automatically randomized) and then update the three elements of the UI accordingly. For example, I could update the *word* portion of a definition as shown in Listing 5:

Listing 5. Updating a UI element programmatically

```
Word aWord = engine.getWord();
TextView wordView = (TextView) findViewById(R.id.word_study_word);
wordView.setText(aWord.getSpelling());
```

The `findViewById` in Listing 5 is an Android platform call that takes an integer ID, which you can get from your app's `R` class. You can programmatically set the text of a `TextView`. You can also set the font type, font color, or size of text display, like so: `wordView.setTextColor(Color.RED)`.

In Listing 6, I follow essentially the same process to update the definition and part-of-speech elements of my app's UI:

Listing 6. More programmatic updates

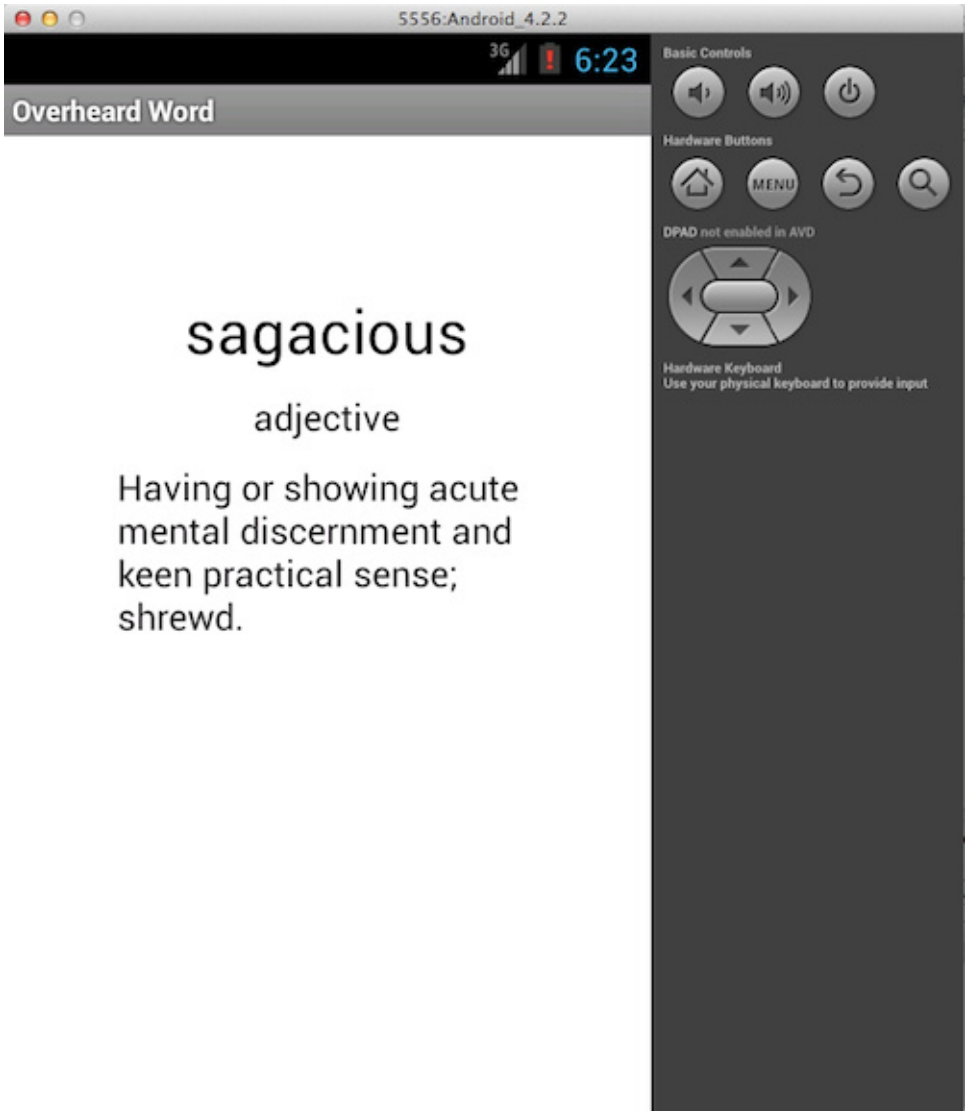
```
Definition firstDef = aWord.getDefinitions().get(0);
TextView wordPartOfSpeechView = (TextView) findViewById(R.id.word_study_part_of_speech);
wordPartOfSpeechView.setText(firstDef.getPartOfSpeech());

TextView defView = (TextView) findViewById(R.id.word_study_definition);
defView.setText(formatDefinition(aWord));
```

Notice in Listing 5 and Listing 6 how useful the `R` file is for referencing layout elements by name. The `formatDefinition` method, which resides in my `Activity` class, takes a definition string and capitalizes its first character. The method also formats the string with a period at the end of the sentence, if one isn't already there. (Note that Thingamajig doesn't have anything to do with formatting — it's just a word engine!)

I'm done updating these UI elements so I'll start up my app and check the results. *Et voilà!* I now have a legitimate word to study!

Figure 4. Overheard Word has words!



Adding gestures: Connecting swipes to words

Now that I can effectively display a word, I want to enable users to swipe quickly through all the words in my word engine. To make my life easier, I'm going to use

Gesticulate, my own third-party library that calculates swipe speed and direction. I also put the swipe logic into a method dubbed `initializeGestures`.

Once I've initialized swipe gestures, my first step is to move the logic for displaying a single word into a new method, which I can invoke in order to display a new word when someone swipes. My updated `onCreate` method (which is initially invoked when Android creates an app instance) is shown in Listing 7:

Listing 7. onCreate now displays a word while initializing gestures

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  Log.d(APP, "onCreated Invoked");
  setContentView(R.layout.activity_overheard_word);

  initializeGestures();

  List<Word> words = buildWordList();

  if (engine == null) {
    engine = WordStudyEngine.getInstance(words);
  }
  Word firstWord = engine.getWord();
  displayWord(firstWord);
}
```

Notice the `engine` variable, which I've defined as a `private static` member variable of the `OverheardWord Activity` itself. I'll explain why I did it this way shortly.

Next, I go into the `initGestureDetector` method, which if you are following along by looking at the code you cloned, is referenced in the `initializeGestures` method. If you remember, the `initGestureDetector` method has the logic to perform an action when a user swipes up, down, left, or right on the device screen. In Overheard Word, I want to display a new word when a user swipes from right to left (which is a *left swipe*). I start by removing the `Toast` message that was my placeholder for that code, replacing it with a call to `displayWord`, shown in Listing 8:

Listing 8. initGestureDetector now displays a word when swiping left

```
private GestureDetector initGestureDetector() {
  return new GestureDetector(new SimpleOnGestureListener() {
    public boolean onFling(MotionEvent e1, MotionEvent e2,
      float velocityX, float velocityY) {
      try {
       final SwipeDetector detector = new SwipeDetector(e1, e2, velocityX, velocityY);
       if (detector.isDownSwipe()) {
         return false;
       } else if (detector.isUpSwipe()) {
         return false;
       } else if (detector.isLeftSwipe()) {
         displayWord(engine.getWord());
       } else if (detector.isRightSwipe()) {
         Toast.makeText(getApplicationContext(),
           "Right Swipe", Toast.LENGTH_SHORT).show();
       }
      } catch (Exception e) {}
     return false;
```

```
    }
  });
}
```

My word engine, represented by the `engine` variable, needs to be accessible throughout the `Activity`. That's why I defined it as a member variable, ensuring that a new word is displayed every time there is a left swipe.

Swiping back and forth

Now when I open my app and start swiping, I see a new word and definition displayed whenever I swipe to the left. But when I swipe in the other direction I just get a tiny message informing me that I've swiped. Wouldn't it be better if I was able to right-swipe back to the previous word?

Going backwards doesn't seem like it should be difficult. Maybe I could use a *stack* and just pop off the top element that was placed there by the previous left swipe? That idea falls apart when a user left-swipes after having gone back, though. If the top position was popped off, a *new* word would be displayed rather than the one the user previously saw. Thinking that through, I'm inclined to try a linked-list approach, where words previously viewed can be picked off as the user swipes back and forth.

I'll start by creating a `LinkedList` of all the words that have been displayed, shown in Listing 9. I'll then keep a pointer to the index of the element in that list that I can use to retrieve words that have been seen. Naturally, I'll make these `static` member variables. I'll also initialize my pointer to `-1` and increment it whenever I add a new word to the `LinkedList` instance. Like most array-like backed collections in any language, `LinkedList` is 0-indexed.

Listing 9. New member variables

```
private static LinkedList<Word> wordsViewed;
private static int viewPosition = -1;
```

In Listing 10, I initialize the `LinkedList` in the `onCreate` method of my app:

Listing 10. Initializing a LinkedList

```
if (wordsViewed == null) {
  wordsViewed = new LinkedList<Word>();
}
```

Now, when I display the first word, I need to add it to the `LinkedList` instance (`wordsViewed`) and increment the pointer variable, `viewPosition`, shown in Listing 11:

Listing 11. Don't forget to increment the view position

```
Word firstWord = engine.getWord();
wordsViewed.add(firstWord);
viewPosition++;
displayWord(firstWord);
```

The logic of swipe

Now I'm into the bulk of the logic, which requires some thinking. When a user swipes left, I want to show the next word and when he or she swipes right, I want to show the previous one. If the user swipes right again, then the second previous word should be shown, and so forth. This should continue until the user arrives back at the first displayed word. If the user then starts swiping left again, the list of words should appear in the same order as they did previously. The last bit is tricky because the default implementation of my `WordStudyEngine` is to return words in a *random* order.

In Listing 12, I think I've nailed the logic of what I want to do. The first boolean evaluation is encapsulated in a method called `listSizeAndPositionEql`, which is simply: `wordsViewed.size() == (viewPosition + 1)`.

If `listSizeAndPositionEql` evaluates to *true*, then the app displays a new word via the `displayWord` method. If, however, `listSizeAndPositionEql` is false, then the user must have swiped backwards; thus, the `viewPosition` pointer is used to retrieve the corresponding word from the list.

Listing 12. A LinkedList for scrolling back and forth

```
public boolean onFling(MotionEvent e1, MotionEvent e2, float velX, float velY) {
 try {
  final SwipeDetector detector = new SwipeDetector(e1, e2, velX, velY);
  if (detector.isDownSwipe()) {
   return false;
  } else if (detector.isUpSwipe()) {
   return false;
  } else if (detector.isLeftSwipe()) {
    if (listSizeAndPositionEql()) {
     viewPosition++;
     Word wrd = engine.getWord();
     wordsViewed.add(wrd);
     displayWord(wrd);
    } else if (wordsViewed.size() > (viewPosition + 1)) {
       if (viewPosition == -1) {
        viewPosition++;
       }
      displayWord(wordsViewed.get(++viewPosition));
    } else {
      return false;
   }
  } else if (detector.isRightSwipe()) {
    if (wordsViewed.size() > 0 && (listSizeAndPositionEql() || (viewPosition >= 0))) {
      displayWord(wordsViewed.get(--viewPosition));
    } else {
      return false;
    }
  }
 } catch (Exception e) {}
 return false;
}
```

Note that if `viewPosition` is `-1`, then the user has swiped backwards all the way back to the beginning — in which case the pointer into the list is incremented back to 0. This is because there is no `-1` element in a Java-based `LinkedList` implementation.

(In some other languages, negative positional values start from the rear of the list, so `-1` would be the tail element.)

Handling right-swipe logic is fairly easy once you've grasped what's going on with left swipes. Basically, if there's a word in the `wordsViewed` instance, then you need to use a decremented value of the pointer to access that previously viewed word.

Give it a try: Start up an instance of Overheard Word and swipe back and forth to study some words. Each time you swipe, the UI will be updated appropriately.

In conclusion

Overheard Word is coming along nicely now that we've started to add some more interesting features on top of the basic Android shell. There's more to do, of course, but we now have a functioning app that handles swipes, has icons and menus, and even reads from a custom, third-party word file. Next month, I'll show you how to add more styling to Overheard Word, then implement a clever quiz feature.

Resources

Learn

- **Previous articles in this series**:
  - "A gentle introduction to Android" (March 2013)
  - "Take a swipe at it! Programming gestures in Android" (April 2013)
  - "Activities and icons in your Android application lifecycle" (May 2013)
- "Working with XML on Android" (Michael Galpin, developerWorks, June 2009): More hands-on learning about Android! Use XML to parse an RSS feed into a list of simple Java objects for an Android `ListView`.

Get products and technologies

- Overheard Word: An Android demo application in progress, hosted on GitHub.
- Thingamajig: A simple word-definition engine used for Overheard Word.
- Gesticulate: Does the math of Android swipe detection for you.
- Download Android: The Android SDK provides you the API libraries and developer tools necessary to build, test, and debug apps for Android.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Andrew Glover

Andrew Glover is a developer and mobile technology enthusiast. He's currently the CTO of App47 and is the founder of the easyb behavior-driven development (BDD) framework. He is also the co-author of three books: *Continuous Integration*, *Groovy in Action*, and *Java Testing Patterns*. You can keep up with Andrew by reading his blog and by following him on Twitter.