15

# Pizza Ordering App

## Text-to-Speech, Speech-to-Text and Telephony

## Objectives

In this chapter you'll:

- Use Android's text-to-speech engine to speak audio instructions to the user.
- Use Android's speech-to-text engine to interpret voice input from the user.
- Use the `SMSManager` to send text messages.
- Send `Message` objects to a `Handler` to ensure that GUI modifications occur in the GUI thread.

Outline

**15.1** Introduction
**15.2** Test-Driving the **Pizza** Ordering App
**15.3** Technologies Overview
**15.4** GUI and Resource Files
15.4.1 Creating the Project
15.4.2 `AndroidManifest.xml`
15.4.3 `main.xml`, `strings.xml` and `arrays.xml`
**15.5** Building the App
**15.6** Wrap-Up

## 15.1 Introduction

The **Pizza** ordering app (Fig. 15.1) uses Android's *text-to-speech* and *speech-to-text engines* to communicate with the user by speaking text and by receiving the user's spoken input. The app creates a pizza order by asking the user to answer questions about the pizza size and toppings. The user responds by speaking the answer into the phone when prompted. If the app cannot understand the user or gets an unexpected answer, the app asks the user to repeat the answer. After processing the user's responses, the app summarizes the order, asks the user whether it's correct and whether it should be submitted. If so, the app sends the order to a mobile phone number (specified in the app's `strings.xml` file) as an SMS message using the Android telephony APIs. If the user wishes to change the order, the app resets and begins asking the questions again. After the order is placed, the user has the option to exit the app or begin again with a new order.

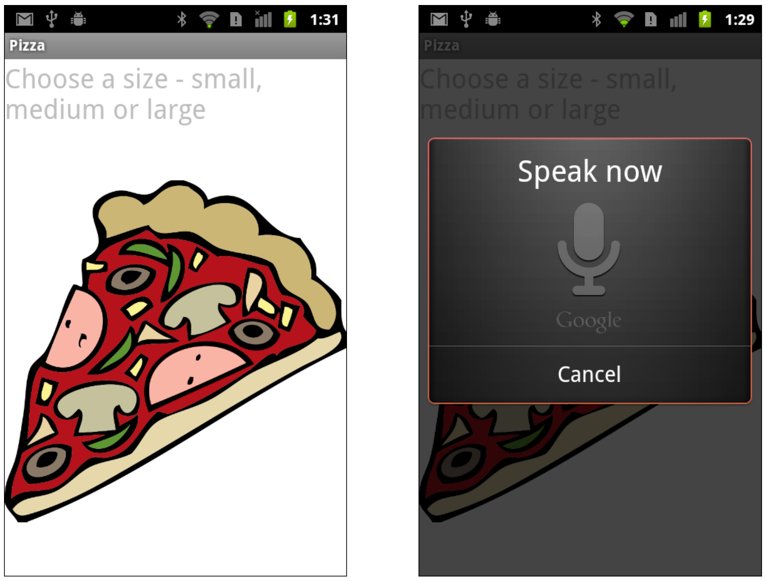
1:31
Pizza
Choose a size - small, medium or large
1:29
Pizza
Choose a size - small, medium or large
Speak now
Google
Cancel

**Fig. 15.1** | **Pizza** ordering app.

## 15.2 Test-Driving the Pizza Ordering App

*Opening and Running the App*

Open Eclipse and import the **Pizza** app project. To import the project:

1. Select **File > Import…** to display the **Import** dialog.
2. Expand the **General** node, select **Existing Projects into Workspace**, then click **Next >**.
3. To the right of the **Select root directory:** text field, click **Browse…**, then locate and select the **Pizza** folder.
4. Click **Finish** to import the project.

At the time of this writing, the speech synthesis and speech recognition capabilities and the SMS message-sending capability work only on actual devices, not in the Android emulator. In addition, a network connection is required (data plan or WiFi) for the voice recognition to work. To use the SMS message-sending functionality, enter your own mobile phone number for the `phone_number` `<string>` resource in `strings.xml`. Ensure that you have an Android device with USB debugging enabled connected to your computer, right click the project's folder and select **Run As > Android Application** to install and run the app on your device.

### *Choosing your Pizza*
Listen to each question spoken by the app—the questions are also displayed on the screen for your convenience. Respond to each question only after the app prompts you to speak. Be sure to speak clearly into your device's microphone. If there's too much background noise the app may ask you to repeat certain answers.

### *Sending an Order*
The app will repeat your completed order back to you, then ask if you want to place the order. Say "yes" to submit the order, which sends an SMS message to the phone number specified in your `strings.xml` file. If the phone number specified represents an actual mobile phone, that phone will receive an SMS text message detailing your order; otherwise, the SMS message will not send correctly.

## 15.3 Technologies Overview

### *Speech Synthesis*
The app speaks to the user using an instance of the **`TextToSpeech`** class. The text-to-speech engine requires initialization that's performed asynchronously. For this reason, the app's **`TextToSpeech.OnInitListener`** is notified when this initialization completes. `TextToSpeech`'s **`speak`** method converts `String`s to audio messages. A **`TextToSpeech.OnUtteranceCompletedListener`** is notified when the speech synthesizer finishes speaking an audio message.

### *Speech Recognition*
The app listens for user input by launching an `Intent` for the **`RecognizerIntent`** using the **`RecognizerIntent.ACTION_RECOGNIZE_SPEECH`** constant. We use `startActivityForResult` to receive the speech recognition results in `Activity`'s `onActivityResult` method. An `ArrayList` of possible matches for the user's speech is included as an *extra* in the `Intent` returned by the `RecognizerIntent` and passed to `onActivityResult`. By comparing the elements in this `ArrayList` to options in the ordering menu we can determine which option the user chose and build the order accordingly.

*Sending SMS Messages*
When an order is completed, the app sends a text message programmatically using class **`SMSManager`**. `SMSManager`'s `static` method **`getDefault`** returns the `SMSManager` object that your app can use to send a message. `SMSManager` method **`sendTextMessage`** sends an SMS message to a specified phone number. One of the arguments to `sendTextMessage` method is a `PendingIntent` that is broadcast when the SMS message is sent. This enables us to use a **`BroadcastReceiver`** to listen for the broadcast to determine whether the SMS message was sent successfully.

*Using a **`Handler`** to Pass **`Messsage`**s Between Threads*
As you know, all GUI modifications must be performed from the GUI thread of execution in Android. In this app, other non-GUI threads need to notify the GUI thread to display text. For example, speech synthesis happens in a separate thread of execution. When speech synthesis completes and we need to display text, we'll notify the GUI thread by passing a **`Message`** object to a `Handler` that's created from the GUI thread. A `Handler`'s **`handleMessage`** method is called on the thread that created the `Handler`.

## 15.4 GUI and Resource Files

In this section, we create the **Pizza** ordering app and discuss its XML files.

### 15.4.1 Creating the Project

Begin by creating a new Android project named `Pizza`. Specify the following values in the **New Android Project** dialog, then press **Finish**:

- **Build Target:** Ensure that **Android 2.3.3** is checked
- **Application name:** `Pizza`
- **Package name:** `com.deitel.pizza`
- **Create Activity:** `Pizza`
- **Min SDK Version:** `8`

### 15.4.2 `AndroidManifest.xml`

Figure 15.2 shows this app's `AndroidManifest.xml` file. The only new feature is the permission **`android.permission.SEND_SMS`** for sending SMS messages (line 16).

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.deitel.pizza" android:versionCode="1"
4     android:versionName="1.0">
5     <application android:icon="@drawable/icon"
6        android:label="@string/app_name" android:debuggable="true">
7        <activity android:name=".Pizza" android:screenOrientation="portrait"
8           android:label="@string/app_name">
9           <intent-filter>
10             <action android:name="android.intent.action.MAIN" />
```

**Fig. 15.2** | `AndroidManifest.xml`. (Part 1 of 2.)

```
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
   <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="10"/>
   <uses-permission android:name="android.permission.SEND_SMS"/>
</manifest>
```

**Fig. 15.2** | `AndroidManifest.xml`. (Part 2 of 2.)

### 15.4.3 `main.xml`, `strings.xml` and `arrays.xml`

The `main.xml` layout for this app is a vertical `LinearLayout` containing a `TextView` and an `ImageView`. We display the spoken `Strings` in the `TextView` so that the user can also read them. The app's `Strings` are defined as `<string>` resources in `strings.xml` and as `<string-array>` resources in `arrays.xml`. You can review the contents of these XML files by opening them in Eclipse.

## 15.5 Building the App

The `Pizza` class (Figs. 15.3–15.17) is the only `Activity` in the app. The app asks a number of questions to determine the user's desired pizza order, then sends the order as an SMS message to a phone number that's specified as a `<string>` resource in `strings.xml`.

#### *`Pizza` `Activity` Class `package` Statement, `import` Statements and Fields*

Figure 15.3 contains the `package` statement, `import` statements and fields for class `Pizza`. We've highlighted the `import` statements for the new classes and interfaces that were introduced in Section 15.3. We discuss the class's fields as they're used. Method `loadResources` (Fig. 15.7) initializes most of the class's instance variables using XML resources that we load from `strings.xml` and `arrays.xml`.

```
// Pizza.java
// Main Activity for the Pizza App.
package com.deitel.pizza;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Locale;

import android.app.Activity;
import android.app.PendingIntent;
import android.content.ActivityNotFoundException;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.res.Resources;
import android.os.Bundle;
import android.os.Handler;
```

**Fig. 15.3** | `Pizza` `Activity` class `package` statement, `import` statements and fields. (Part 1 of 3.)

```
import android.os.Message;
import android.speech.RecognizerIntent;
import android.speech.tts.TextToSpeech;
import android.speech.tts.TextToSpeech.OnInitListener;
import android.speech.tts.TextToSpeech.OnUtteranceCompletedListener;
import android.telephony.SmsManager;
import android.widget.TextView;
import android.widget.Toast;

public class Pizza extends Activity
{
   private String phoneNumber; // phone number to which order is sent

   // identifying String for sent SMS message broadcast Intent
   private static final String BROADCAST_STRING =
      "com.deitel.pizza.sent_sms";

   // SMS message broadcast Intent
   private BroadcastReceiver textMessageStatusBroadcastReceiver;

   // 0-based index of each pizza question
   private static final int PIZZA_SIZE_INDEX = 1;
   private static final int PEPPERONI_INDEX = 2;
   private static final int MUSHROOM_INDEX = 3;
   private static final int ORDER_SUMMARY_INDEX = 4;

   // message IDs to differentiate between a
   // regular message and the final message
   private final static int UPDATE_TEXT_ID = 15;
   private final static int FINAL_UPDATE_TEXT_ID = 16;
   private final static int DISPLAY_TOAST_ID = 17;

   // String identifiers for restoring instance state
   private final static String INDEX_ID = "index";
   private final static String ORDER_ID = "order";
   private final static String LISTENING_ID = "listening";

   private TextToSpeech textToSpeech; // converts text to speech
   private int currentMessageIndex; // index of the current message

   private boolean waitingForResponse; // waiting for user response?
   private boolean listening; // waiting for Activity result?
   private TextView messageText; // used to display the current message
   private String order; // the pizza order

   private String[] audioMessages; // messages spoken by the app
   private String[] displayMessages; // messages displayed by the app

   private String errorMessageString; // message for unexpected response
   private String finalMessageString; // message when app sends order

   // possible choices for each of the five order options
   private String[][] choices = new String[6][];
```

**Fig. 15.3** | Pizza Activity class package statement, import statements and fields. (Part 2 of 3.)

```
72
73    private String positiveResponseString; // "Yes"
74    private String negativeResponseString; // "No"
75
76    private Resources resources; // used to access the app's Resources
77    private boolean quitInProgress;
78
79    private HashMap<String, String> ttsParams; // TextToSpeech parameters
80
```

**Fig. 15.3** | `Pizza Activity` class `package` statement, `import` statements and fields. (Part 3 of 3.)

### *Overriding* `Activity` *Method* `onCreate`

The `onCreate` method (Fig. 15.4) sets up the `Pizza Activity`. Lines 89–115 create a new `TextToSpeech` object and configure its listeners. We'll use this object to speak commands and questions to the user during the pizza-ordering process. The first argument to the `TextToSpeech` constructor is the `Context` in which the object will be used. The second argument is the `TextToSpeech.OnInitListener` (lines 90–114) that's notified when the `TextToSpeech` engine's initialization is complete.

```
81    // Called when the Activity is first created
82    @Override
83    public void onCreate(Bundle savedInstanceState)
84    {
85       super.onCreate(savedInstanceState);
86       setContentView(R.layout.main); // set the Activity's layout
87
88       // initialize TextToSpeech engine and register its OnInitListener
89       textToSpeech = new TextToSpeech(this,
90          new OnInitListener()
91          {
92             // called when the TextToSpeech is initialized
93             @Override
94             public void onInit(int status)
95             {
96                // speak U.S. English
97                textToSpeech.setLanguage(Locale.US);
98
99                // set listener that responds to events generated
100               // when messages are completed
101               textToSpeech.setOnUtteranceCompletedListener(
102                  new OnUtteranceCompletedListener()
103                  {
104                     @Override
105                     public void onUtteranceCompleted(String id)
106                     {
107                        utteranceCompleted();
108                     } // end method onUtteranceCompleted
109                  } // end anonymous inner class
110               ); // end call to setOnUtteranceCompletedListener
```

**Fig. 15.4** | Overriding `Activity` method `onCreate`. (Part 1 of 2.)

```
111
112               playFirstMessage();
113            } // end method onInit
114         } // end anonymous inner class that implements OnInitListener
115      ); // end call to TextToSpeech constructor
116
117      // used in calls to TextToSpeech's speak method to ensure that
118      // OnUtteranceCompletedListener is notified when speech completes
119      ttsParams = new HashMap<String, String>();
120      ttsParams.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, "speak");
121
122      currentMessageIndex = 1; // start at the first message
123      waitingForResponse = false; // not waiting for user response
124
125      // get the Activity's TextView
126      messageText = (TextView) findViewById(R.id.mainText);
127
128      loadResources(); // load String resources from xml
129   } // end method onCreate
130
```

**Fig. 15.4** | Overriding `Activity` method `onCreate`. (Part 2 of 2.)

The `TextToSpeech.OnInitListener`'s **`onInit` method** is called when the `TextToSpeech` object finishes initializing. Line 97 uses `TextToSpeech`'s **`setLanguage` method** to specify that the app will speak U.S. English (`Locale.US`). Class `Locale` provides constants for many locales, but it's not guaranteed that all are supported on every device. You can use method **`isLanguageAvailable`** to check whether a specific `Locale` is available before using it. Lines 101–110 define the `TextToSpeech` object's `OnUtteranceCompletedListener`, which is notified when the `TextToSpeech` object finishes speaking a message. When this occurs, the event handler's **`onUtteranceCompleted` method** (lines 104–108) calls our method `utteranceCompleted` (Fig. 15.9) to process that event.

Lines 119–120 create and configure the `ttsParams` `HashMap` that will be used as the last argument in each call to the `TextToSpeech` object's `speak` method. To ensure that the `OnUtteranceCompletedListener` is notified when speech completes, the `HashMap` must contain the key `TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID` with a value that's a non-empty string. The value associated with this key is passed to the `OnUtteranceCompletedListener`'s `onUtteranceCompleted` method and can be used in the method to determine the text that the TTS engine just completed speaking, so that you can perform specific tasks based on that information. We do not use the `onUtteranceCompleted` method's argument in this app.

Instance variable `currentMessageIndex` (line 122) keeps track of the index in a `String` array of the messages and questions the app speaks to the user. The `waitingForResponse` `boolean` indicates whether or not the app is currently waiting for the user to respond before continuing with the order—the app has not spoken any text yet, so this is initialized to `false` (line 123). Line 128 calls our method `loadResources` (Fig. 15.7) to load the `String` values from the app's `strings.xml` and `arrays.xml` files.

### *Overriding* `Activity` *Method* `onResume`

When the user completes the order, the app asks whether the order should be sent as an SMS message. To ensure that the SMS is sent, we can register a `BroadcastReceiver` to check the result of the `Intent` that sent the message. Method `onResume` (Fig. 15.5) creates and registers the `textMessageStatusBroadcastReceiver`. When the `BroadcastReceiver`'s `onReceive` method is called, we check whether the result code is not `Activity.RESULT_OK` (line 144), in which case we display an error message on the app. The `BroadcastReceiver` is notified asynchronously, so we need to display the error from the GUI thread, which we do by passing a `Message` to a `Handler`'s **`sendMessage` method** (lines 146–148). The `viewUpdateHandler` is defined in Fig. 15.15 and used throughout the `Pizza` `Activity`.

```
131    // called when this Activity is resumed
132    @Override
133    public void onResume()
134    {
135       super.onResume();
136
137       // create BroadcastReceiver to receive SMS message status broadcast
138       textMessageStatusBroadcastReceiver = new BroadcastReceiver()
139       {
140          @Override
141          public void onReceive(Context context, Intent intent)
142          {
143             // if the message was not sent
144             if (getResultCode() != Activity.RESULT_OK)
145             {
146                viewUpdateHandler.sendMessage(
147                   viewUpdateHandler.obtainMessage(Pizza.DISPLAY_TOAST_ID,
148                      R.string.text_error_message, 0, null));
149             } // end if
150          } // end method onReceive
151       }; // end BroadcastReceiver anonymous inner class
152
153      // register the receiver
154      registerReceiver(textMessageStatusBroadcastReceiver,
155         new IntentFilter(Pizza.BROADCAST_STRING));
156    } // end method onResume
157
```

**Fig. 15.5** | Overriding `Activity` method `onResume`.

A `Handler`'s `handleMessage` method executes in the thread from which the `Handler` was created and receives the `Message` sent by the `Handler`'s `sendMessage` method. Because `viewUpdateHandler` is an instance variable of `Activity` class `Pizza`, the `viewUpdateHandler` is created in the GUI thread of execution. This helps us ensure that modifications to the GUI happen in the GUI thread.

Android maintains a *global pool* of reusable `Message` objects, so rather than creating `Message` objects with the default constructor, lines 147–148 create the `Message` that's passed to the `viewUpdateHandler` by calling `Handler` method `obtainMessage`. The version of `obtainMessage` used here requires four arguments—an `int` ID that indicates the

Message's purpose (used to decide how to process it) and two arbitrary int values and an arbitrary Object that can be used when handling the Message. In our case, the second argument is a String resource ID for the error message we'll display. We pass 0 and null for the last two arguments because we do not use them in this app.

Lines 154–155 pass the BroadcastReceiver and a new **IntentFilter** to Activity's **registerReceiver method** to allow the app to receive broadcast Intents. The String argument to the IntentFilter constructor is an app-specific String that allows the app to receive the broadcasts intended for the app. When we send the SMS message (Fig. 15.14), we'll arrange to receive a broadcast Intent with an action String that uses the same Pizza.BROADCAST_STRING constant.

### *Overriding* Activity *Method* onPause

When the Activity is paused, there's no need to receive broadcast Intents, so we override onPause (Fig. 15.6) to unregister our BroadcastReceiver by passing it to Activity's **unregisterReceiver method**.

```java
   // called when this Activity is paused
   @Override
   public void onPause()
   {
      super.onPause();

      // if the BroadcastReceiver is not null, unregister it
      if (textMessageStatusBroadcastReceiver != null)
         unregisterReceiver(textMessageStatusBroadcastReceiver);

      textMessageStatusBroadcastReceiver = null;
   } // end method onPause

```

**Fig. 15.6** | Overriding Activity method onPause.

### Pizza *Method* loadResources

The loadResources method (Fig. 15.7) is called from onCreate (line 128 of Fig. 15.4) and loads the app's String and String array resources using the Activity's Resource object's getString and getStringArray methods. The choices two-dimensional String array contains the possible answers for each question asked by the app. For example, the String array at index PEPPERONI_INDEX contains all acceptable responses to the question: "Do you want pepperoni?"—in this case, "Yes" and "No". These Strings are loaded in the array binaryChoices (lines 194–195) and reused for several of the questions.

```java
   // load String resources from XML
   private void loadResources()
   {
      resources = getResources(); // get the app's resources
      phoneNumber = resources.getString(
         R.string.phone_number); // load audio messages
```

**Fig. 15.7** | Pizza method loadResources. (Part 1 of 2.)

```
      audioMessages = resources.getStringArray(
         R.array.audio_messages); // load audio messages
      displayMessages = resources.getStringArray(
         R.array.display_messages); // load the display messages
      errorMessageString = resources.getString(
         R.string.error_message); // error message
      finalMessageString = resources.getString(
         R.string.final_message); // final message
      positiveResponseString = resources.getString(
         R.string.positive_response); // "Yes"
      negativeResponseString = resources.getString(
         R.string.negative_response); // "No"

      // initialize the pizza order
      order = resources.getString(R.string.initial_order);

      // load the valid user responses
      String[] binaryChoices =
         resources.getStringArray(R.array.binary_choices);
      choices[PIZZA_SIZE_INDEX] =
         resources.getStringArray(R.array.size_choices);
      choices[PEPPERONI_INDEX] = binaryChoices;
      choices[MUSHROOM_INDEX] = binaryChoices;
      choices[ORDER_SUMMARY_INDEX] = binaryChoices;
   } // end method loadResources

```

**Fig. 15.7** | `Pizza` method `loadResources`. (Part 2 of 2.)

### *`Pizza` Method `playFirstMessage`*

The `playFirstMessage` method (Fig. 15.8) is called (Fig. 15.4, line 112) after the `TextToSpeech` engine is initialized. The method speaks the app's welcome message (stored in `audioMessages[0]`) by calling `TextToSpeech`'s `speak` method with three arguments—the `String` to speak, the queue mode and a `HashMap` of parameters for the `TextToSpeech` engine. The queue mode is either `TextToSpeech.QUEUE_FLUSH` or `TextToSpeech.QUEUE_ADD`. The mode `QUEUE_FLUSH` empties the speech queue (the list of `String`s waiting to be spoken) so that the new `String` can be spoken immediately. The mode `QUEUE_ADD` adds the new text to speak to the end of the speech queue.

```
   // speak the first message
   private void playFirstMessage()
   {
      // speak the first message
      textToSpeech.speak(
         audioMessages[0], TextToSpeech.QUEUE_FLUSH, ttsParams);
   } // end method playFirstMessage

```

**Fig. 15.8** | `Pizza` method `playFirstMessage`.

### *Pizza Method* `utteranceCompleted`

Method `utteranceCompleted` (Fig. 15.9) is called by the `TextToSpeech` object's `onUtteranceCompleted` event handler (Fig. 15.4, lines 104–108) and whenever the app needs to move to the next message to speak. We first obtain from the `ttsParams` object the value of the key `TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID` so we can determine whether the user has chosen to quit the app (lines 220–225). If so, we **`shutDown`** the `TextToSpeech` engine to release its resources and terminate the app by calling `Activity` method `finish`.

```
211      // utility method called when speech completes and
212      // when it's time to move to the next message
213      private void utteranceCompleted()
214      {
215         // if the TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID
216         // contains "quit" terminate the app
217         String quit =
218            ttsParams.get(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID);
219
220         if (quit.equals("quit")) // check whether user wishes to quit
221         {
222            textToSpeech.shutdown(); // shut down the TextToSpeech
223            finish();
224            return;
225         } // end if
226
227         // allow user to quit
228         if (currentMessageIndex >= displayMessages.length &&
229            !quitInProgress)
230         {
231            allowUserToQuit();
232         } // end if
233         else if (!waitingForResponse) // if we're not waiting for a response
234         {
235            // update the TextView
236            viewUpdateHandler.sendMessage(
237               viewUpdateHandler.obtainMessage(UPDATE_TEXT_ID));
238
239            String words = "";
240
241            // summarize the order
242            if (currentMessageIndex == ORDER_SUMMARY_INDEX)
243            {
244               words = resources.getString(R.string.order_summary_prefix);
245               words += order.substring(order.indexOf(':') + 1);
246            } // end if
247
248            words += audioMessages[currentMessageIndex]; // next message
249            words = words.replace(resources.getString(R.string.pepperoni),
250               resources.getString(R.string.pepperoni_speech));
251            words = words.replace(resources.getString(R.string.pizza),
252               resources.getString(R.string.pizza_speech));
253
```

**Fig. 15.9** | Pizza method `utteranceCompleted`. (Part 1 of 2.)

```
254             // speak the next message
255            textToSpeech.speak(words, TextToSpeech.QUEUE_FLUSH, ttsParams);
256            waitingForResponse = true; // we are waiting for a response
257         } // end if
258         else if (!listening && currentMessageIndex > 0)
259         {
260            listen(); // capture the user's response
261         } // end else if
262      } // end method utteranceCompleted
263
```

**Fig. 15.9** | `Pizza` method `utteranceCompleted`. (Part 2 of 2.)

Next, we determine whether the order has been completed (lines 228–229). If so, we call method `allowUserToQuit` to allow the user to exit the app or start a new order. If we're not waiting for a user response (line 233) we pass a `Message` to the `viewUpdateHandler` so that it can update the `TextView`'s text. Lines 239–252 configure the `String words`, which will contain the `String` representation of the words to speak to the user. If we're on the last of the messages that the app speaks to the user (line 242), lines 244–245 summarize the order. Line 248 appends the current `String` from the `audioMessages` array to `words`. Lines 249–250 replace the words "pepperoni" and "pizza" with strings that allow the `TextToSpeech` engine to speak these words with better pronunciation—such as "pehperohnee" for "pepperoni." Then line 255 speaks the message using `TextToSpeech`'s `speak`. We also set `waitingForResponse` to `true`. If we're waiting for a user response (line 258), we call the `listen` method (Fig. 15.10) to start an `Intent` for the speech recognition `Activity`.

### *`Pizza` Method `listen`*

The `listen` method (Fig. 15.10) uses an `Intent` (270–271) to start an `Activity` that listens for audio input from the user. The `RecognizerIntent.ACTION_RECOGNIZE_SPEECH` constant represents the speech recognition `Activity`. We launch the `Intent` using `startActivityForResult` (line 276) so that we can receive results in the `Pizza Activity`'s overridden `onActivityResult` method. We `catch` an `ActivityNotFoundException` that will be thrown by an AVD or any device that does not have speech recognition capability. If this happens, we send a `Message` to the `viewUpdateHandler` to display a `Toast` explaining why this app will not work.

```
264      // listens for a user response
265      private void listen()
266      {
267         listening = true; // we are now listening
268
269         // create Intent for speech recognition Activity
270         Intent speechRecognitionIntent =
271            new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
272
273         // try to launch speech recognition Activity
274         try
275         {
```

**Fig. 15.10** | `Pizza` method `listen`. (Part 1 of 2.)

```
276             startActivityForResult(speechRecognitionIntent, 0);
277          } // end try
278          catch (ActivityNotFoundException exception)
279          {
280             viewUpdateHandler.sendMessage(viewUpdateHandler.obtainMessage(
281                Pizza.DISPLAY_TOAST_ID, R.string.no_speech_message, 0, null));
282          } // end catch
283       } // end method listen
284
```

**Fig. 15.10** | `Pizza` method `listen`. (Part 2 of 2.)

### *Overriding* `Activity` *Method* `onActivityResult`

The `Pizza` `Activity` overrides the `onActivityResult` method (Fig. 15.11) to process results from the speech recognition `Activity`. We pass the `RecognizerIntent.EXTRA_RESULTS` to the received `Intent`'s `getStringArrayListExtra` (lines 296–298) to get an `ArrayList` containing `String` representations of the speech recognition `Activity`'s interpretations of the user's spoken input. Speech recognition is not exact, so if any of these `String`s matches a response that the app expects, we'll assume that the user spoke that response and act accordingly. Lines 316–327 loop through each of the valid choices, comparing them with each of the possible matches to the user's speech input. We save the first match in `result` (line 323). If there's no match, we call the `playError` method to ask the user to repeat the response (line 330). Otherwise lines 331–418 process the user's response. Lines 331–371 quit or continue the app. Lines 373–387 send the pizza order or start over. Lines 388–412 continue the order process—we call the `utteranceCompleted` method (line 411) with the empty `String` to speak the next message to the user. Lines 414–418 process the case in which the user cancels the speech input.

```
285       // called when the speech recognition Activity returns
286       @Override
287       protected void onActivityResult(int requestCode, int resultCode,
288          Intent dataIntent)
289       {
290          listening = false;
291
292          // if there was no error
293          if (requestCode == 0 && resultCode == RESULT_OK)
294          {
295             // get list of possible matches to user's speech
296             ArrayList<String> possibleMatches =
297                dataIntent.getStringArrayListExtra(
298                   RecognizerIntent.EXTRA_RESULTS);
299
300             // get current list of possible valid choices
301             String[] validResponses;
302
303             if (!quitInProgress)
304                validResponses = choices[currentMessageIndex];
```

**Fig. 15.11** | Overriding `Activity` method `onActivityResult`. (Part 1 of 4.)

```
         else
            validResponses =
               resources.getStringArray(R.array.binary_choices);

         if (validResponses == null)
            return;

         String result = null;

         // for each possible valid choice, compare to the user's speech
         // to determine whether the user spoke one of those choices
         checkForMatch:
         for (String validResponse : validResponses)
         {
            for (String match : possibleMatches)
            {
               if (validResponse.compareToIgnoreCase(match) == 0)
               {
                  result = validResponse; // store the user response
                  break checkForMatch; // stop checking possible responses
               } // end if
            } // end for
         } // end for

         if (result == null) // there was no match
            playError(); // ask the user to repeat the response
         else if (quitInProgress)
         {
            quitInProgress = false;

            // the user said to quit
            if (result.equalsIgnoreCase(positiveResponseString))
            {
               if (currentMessageIndex >= displayMessages.length)
               {
                  reset(); // reset the order
                  return; // return
               } // end if
               else
               {
                  ttsParams.put(
                     TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, "quit");

                  // speak the final message
                  textToSpeech.speak(
                     resources.getString(R.string.quit_message),
                     TextToSpeech.QUEUE_FLUSH, ttsParams);
               } // end else
            } // end if
            else // the user wants to return
            {
               if (currentMessageIndex >= displayMessages.length)
               {
```

**Fig. 15.11** | Overriding `Activity` method `onActivityResult`. (Part 2 of 4.)

```
                     ttsParams.put(
                        TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, "quit");

                     // speak the final message
                     textToSpeech.speak(
                        resources.getString(R.string.leave_message),
                        TextToSpeech.QUEUE_FLUSH, ttsParams);
                  } // end if
                  else
                  {
                     listen();
                  } // end else
               } // end else
            } // end else if
            // there was a match and it is on the last message
            else if (currentMessageIndex == displayMessages.length - 1)
            {
               // the user said to send the order
               if (result.equalsIgnoreCase(positiveResponseString))
               {
                  waitingForResponse = false;
                  ++currentMessageIndex;
                  sendMessage(); // send the order as a text message
               } // end if
               else // the user canceled the order
               {
                  reset(); // reset the order
                  return; // return
               } // end else
            } // end else if
            else // there was a match and it is not the last message
            {
               // the user responded positively
               if (result.equalsIgnoreCase(positiveResponseString))
               {
                  // if previous question asked if the user wants pepperoni
                  if (currentMessageIndex == PEPPERONI_INDEX)
                  {
                     // add pepperoni to the pizza order
                     order += resources.getString(R.string.pepperoni);
                  } // end if
                  else if (currentMessageIndex == MUSHROOM_INDEX)
                  {
                     // add mushrooms to the pizza order
                     order += resources.getString(R.string.mushrooms);
                  } // else if
               } // end if
               else if (!result.equalsIgnoreCase(negativeResponseString))
                  order += ", " + result; // update the order

               waitingForResponse = false;
               ++currentMessageIndex; // move to the next question

```

**Fig. 15.11** | Overriding `Activity` method `onActivityResult`. (Part 3 of 4.)

```
411             utteranceCompleted(); // move to next message
412          } // end else
413       } // end if
414       else if ((currentMessageIndex > 0 && !listening) ||
415          resultCode == Activity.RESULT_CANCELED)
416       {
417          allowUserToQuit(); // listen for user input
418       } // end else
419
420       // call super method
421       super.onActivityResult(requestCode, resultCode, dataIntent);
422    } // end method onActivityResult
423
```

**Fig. 15.11** | Overriding `Activity` method `onActivityResult`. (Part 4 of 4.)

### *`Pizza` Method `playError`*

The `playError` method (Fig. 15.12, lines 425–429) is called by `onActivityResult` whenever `Android`'s speech recognizer fails to recognize the user's spoken response. Lines 427–428 use the `textToSpeech` object's `speak` method to ask the user to try again. Method reset (lines 432–441) is called by `onActivityResult` whenever the user decides to restart the order process.

```
424    // called when the user says an unexpected response
425    private void playError()
426    {
427       textToSpeech.speak(errorMessageString, // play error message
428          TextToSpeech.QUEUE_FLUSH, ttsParams);
429    } // end method playError
430
431    // start a new order
432    private void reset()
433    {
434       // reset the instance variables associated with taking an order
435       currentMessageIndex = 1;
436       order = resources.getString(R.string.initial_order);
437       waitingForResponse = false;
438       listening = false;
439
440       playFirstMessage();
441    } // end method reset
442
```

**Fig. 15.12** | `Pizza` methods `playError` and `reset`.

### *Overriding `Activity` Methods `onSaveInstanceState` and `onRestoreInstanceState`*

`Activity` methods `onSaveInstanceState` and `onRestoreInstanceState` (Fig. 15.13) save and restore the values for the `Pizza` `Activity`'s `currentMessageIndex`, `order` and `listening` instance variables in the event that the `Pizza` `Activity` is pushed to the background and brought back to the foreground.

```
   // save the order state
   @Override
   public void onSaveInstanceState(Bundle savedStateBundle)
   {
      // store the currentMessageIndex, order and listening values
      savedStateBundle.putInt(INDEX_ID, currentMessageIndex);
      savedStateBundle.putString(ORDER_ID, order);
      savedStateBundle.putBoolean(LISTENING_ID, listening);

      super.onSaveInstanceState(savedStateBundle);
   } // end method onSaveInstanceState

   // restore the order state
   @Override
   public void onRestoreInstanceState(Bundle savedStateBundle)
   {
      // retrieve the currentMessageIndex, order and listening values
      currentMessageIndex = savedStateBundle.getInt(INDEX_ID);
      order = savedStateBundle.getString(ORDER_ID);
      listening = savedStateBundle.getBoolean(LISTENING_ID);
      super.onRestoreInstanceState(savedStateBundle);
   } // end method onRestoreInstanceState

```

**Fig. 15.13** | Overriding `Activity` methods `onSaveInstanceState` and `onRestoreInstanceState`.

### *Pizza Method* `sendMessage`

The `sendMessage` method (Fig. 15.14) is called by `onActivityResult` to send the final order `String` as an SMS text message. To do this, we create a new `Intent` (line 469) with an action `String` that matches the one we used to register the `textMessageStatusBroadcastReceiver`. We then use this `Intent` to create a `PendingIntent` (lines 470–471) by calling `PendingIntent`'s `static` **`getBroadcast` method**. Recall from Chapter 14 that a `PendingIntent` represents an `Intent` and an action to perform with that `Intent`. When the `PendingIntent` completes, it broadcasts the `Intent` specified as the third argument to `getBroadcast`—this is the `Intent` that the `BroadcastReceiver` (Fig. 15.5) receives indicating whether the SMS message was sent successfully.

```
466    // send order as a text message
467    private void sendMessage()
468    {
469       Intent broadcastIntent = new Intent(Pizza.BROADCAST_STRING);
470       PendingIntent messageSentPendingIntent =
471          PendingIntent.getBroadcast(this, 0, broadcastIntent, 0);
472
473       // get the default SMSManager
474       SmsManager smsManager = SmsManager.getDefault();
475
```

**Fig. 15.14** | `Pizza` method `sendMessage`. (Part 1 of 2.)

```
476       // send the order to PHONE_NUMBER
477       smsManager.sendTextMessage(phoneNumber, null, order,
478          messageSentPendingIntent, null);
479
480       // display the final message
481       viewUpdateHandler.sendMessage(
482          viewUpdateHandler.obtainMessage(FINAL_UPDATE_TEXT_ID));
483    } // end method sendMessage
484
```

**Fig. 15.14** | `Pizza` method `sendMessage`. (Part 2 of 2.)

Line 474 gets the `SMSManager` by calling `SMSManager static` method `getDefault`. `SMSManager`'s `sendTextMessage` method (lines 477–478) sends the SMS message. The first argument is the phone number to which the message will be sent. The second argument, `null`, indicates that the default SMS center (SMSC) should be used to forward the SMS message to its destination. The third argument is the message to send. The `PendingIntent` in the fourth argument is broadcast when the message is sent—the `PendingIntent`'s result code will indicate whether the sending the SMS succeeded or failed. The last argument (if not `null`) is another `PendingIntent` that's broadcast when the SMS message is delivered to the recipient. Lines 481–482 send a `Message` to the `viewUpdateHandler` to display an order-completed message to the user and to speak that message.

### *`viewUpdateHandler` for Updating the GUI*

The `viewUpdateHandler` (Fig. 15.15) is called throughout the `Pizza Activity` to update the GUI based on the current order state and to display error messages. Lines 489–519 override `Handler`'s `handleMessage` method, which receives a `Message` as an argument and updates the GUI based on the contents of that `Message`. Lines 492–518 process the `Message` based on the ID contained in `receivedMessage.what`. For `Pizza.UPDATE_TEXT_ID`, we display the next message in `displayMessages`, so that the user can see the same text that the app is speaking. For `Pizza.FINAL_UPDATE_TEXT_ID`, we display and speak the `finalMessageString`. For `Pizza.DISPLAY_TOAST_ID`, we display a `Toast` containing the value that was stored in the `Message`'s `arg1` instance variable when the `Message` was sent—this instance variable contains the text to display in the `Toast`.

```
485    // updates the UI
486    private Handler viewUpdateHandler = new Handler()
487    {
488       // displays the given next message
489       public void handleMessage(Message receivedMessage)
490       {
491          // process Message based on the ID stored in receivedMessage.what
492          switch (receivedMessage.what)
493          {
494             case Pizza.UPDATE_TEXT_ID: // if it is not the last message
495                // display the message
496                String text = "";
497
```

**Fig. 15.15** | `viewUpdateHandler` for updating the GUI. (Part 1 of 2.)

```
               // if next message is the last one
               if (currentMessageIndex == displayMessages.length - 1)
                  text = order;

               text += displayMessages[currentMessageIndex];
               messageText.setText(text);
               break;
            case Pizza.FINAL_UPDATE_TEXT_ID: // if order is complete
               // display and play the final message
               messageText.setText(finalMessageString);

               // speak the final message
               textToSpeech.speak(finalMessageString,
                  TextToSpeech.QUEUE_FLUSH, ttsParams);
               break;
            case DISPLAY_TOAST_ID:
               // if speech recognition is not available on this device
               // inform the user using a Toast
               Toast.makeText(Pizza.this, receivedMessage.arg1,
                  Toast.LENGTH_LONG).show();
         } // end switch statement
      } // end method handleMessage
   }; // end Handler

```

**Fig. 15.15** | `viewUpdateHandler` for updating the GUI. (Part 2 of 2.)

### Pizza *Method* `allowUserToQuit`

The `allowUserToQuit` method (Fig. 15.16) is called from the `utteranceCompleted` and `onActivityResult` methods to ask the user whether to exit the **Pizza** app. If we've completed an order (line 529), we ask the user whether to quit the app or to start another order (lines 531–533); otherwise, we ask whether they want to quit or continue the current order.

```
522    // allow the user to exit the app
523    private void allowUserToQuit()
524    {
525       quitInProgress = true;
526       waitingForResponse = true;
527
528       // if the order is complete, ask whether to quit or start new order
529       if (currentMessageIndex >= displayMessages.length)
530       {
531          textToSpeech.speak(
532             resources.getString(R.string.leave_question),
533             TextToSpeech.QUEUE_FLUSH, ttsParams);
534       } // end if
535       else // ask whether to quit or continue order
536       {
```

**Fig. 15.16** | `Pizza` method `allowUserToQuit`. (Part 1 of 2.)

```
            textToSpeech.speak(
               resources.getString(R.string.quit_question),
               TextToSpeech.QUEUE_FLUSH, ttsParams);
         } // end else
      } // end method allowUserToQuit

```

**Fig. 15.16** | `Pizza` method `allowUserToQuit`. (Part 2 of 2.)

#### *Overriding* `Activity` *Method* `onDestroy`

The `onDestroy` method (Fig. 15.17) is called when this `Activity` is destroyed. We call `TextToSpeech`'s `shutdown` method to release the native Android resources used by the `TextToSpeech` engine.

```
   // when the app is shut down
   @Override
   public void onDestroy()
   {
      super.onDestroy(); // call super method
      textToSpeech.shutdown(); // shut down the TextToSpeech
   } // end method onDestroy
} // end class Pizza
```

**Fig. 15.17** | Overriding `Activity` method `onDestroy`.

## 15.6 Wrap-Up

The **Pizza** ordering app used Android's *text-to-speech* and *speech-to-text engines* to communicate with the user by speaking text and by receiving the user's spoken input. Once an order was complete, the app sent the order to a mobile phone number as an SMS message using the Android telephony APIs.

The app used a `TextToSpeech` object to speak text. Because the text-to-speech engine is initialized asynchronously, you used a `TextToSpeech.OnInitListener` so the app could be notified when the initialization completed. You converted text to spoken messages by calling `TextToSpeech`'s `speak` method and determined how to proceed in the app when speech completed by implementing a `TextToSpeech.OnUtteranceCompletedListener`.

You listened for user input by launching a `RecognizerIntent` with the constant `ACTION_RECOGNIZE_SPEECH` then responded to the speech recognition results in the `Pizza` `Activity`'s `onActivityResult` method. The `RecognizerIntent` returned an `ArrayList` of possible matches for the user's speech. By comparing the elements in this `ArrayList` to the app's ordering options, you determined which option the user chose and processed the order accordingly.

When an order was completed, you sent an SMS message programmatically with an `SMSManager` that you obtained with `SMSManager`'s `static` method `getDefault`. You sent the SMS by calling `SMSManager`'s `sendTextMessage` method. You used a `PendingIntent` to receive a notification of whether the SMS message was sent successfully and handled the notification with a `BroadcastReceiver`.

To ensure that all GUI modifications were performed from the GUI thread of execution, you passed a `Message` object to a `Handler` that was created from the GUI thread. The `Handler`'s `handleMessage` method was called on the thread that created the `Handler`—the GUI thread in this app.

In Chapter 16, we present the **Voice Recorder** app, which allows the user to record sounds using the phone's microphone and save the audio files for playback later.