CHAPTER 4

No Texting While Driving

This chapter walks you through the creation of No Texting While Driving, a "text answering machine" app that auto-responds to text messages you receive while you're driving (or in the office, etc.), speaks text messages aloud, and even sends location information as part of the automated text reply. The app demonstrates how you can control some of the great features of an Android phone, including SMS texting, text-to-speech, persistent data, and GPS location sensing.

In January 2010, the United States National Safety Council (NSC) announced the results of a study that found that at least 28 percent of all traffic accidents—close to 1.6 million crashes every year—are caused by drivers using cell phones, and at least 200,000 of those accidents occurred while drivers were texting.[1] As a result, many states have banned drivers from using cell phones altogether.

1 *http://bit.ly/1qiH7aZ*

Daniel Finnegan, a student at the University of San Francisco taking an App Inventor programming class, came up with an app idea to help with the driving and texting epidemic. The app he created, which is shown in *Figure 4-1*, responds automatically (and hands-free) to any text with a message such as "I'm driving right now, I'll contact you shortly."

The app was later extended so that it would speak the incoming texts aloud and add the driver's GPS location to the auto-response text, and it was turned into a tutorial for the App Inventor site.

Some weeks after the app was posted on the App Inventor site, State Farm Insurance created an Android app called On the Move, which had similar functionality to *No Texting While Driving*.

Figure 4-1. The No Texting While Driving app

We don't know if Daniel's app or the tutorial on the App Inventor site influenced On the Move, but it's interesting to consider the possibility that an app created in a beginning programming course (by a creative writing student, no less!) might have inspired this mass-produced piece of software, or at least contributed to the ecosystem that brought it about. It certainly demonstrated how App Inventor has lowered the barrier of entry so that anyone with a good idea can quickly and inexpensively turn his idea into a tangible, interactive app. Clive Thompson of *Wired* magazine picked up on the novelty and wrote this:

> Software, after all, affects almost everything we do. Pick any major problem—global warming, health care, or, in Finnegan's case, highway safety—and clever software is part of the solution. Yet only a tiny chunk of people ever consider learning to write code, which means we're not tapping the creativity of a big chunk of society.[2]

App Inventor is about tapping the creativity Thompson mentions, about opening up the world of software creation to everyone.

What You'll Learn

This is a more complex app than those in the previous chapters, so you'll build it one piece of functionality at a time, starting with the auto-response message. You'll learn about:

2 Clive Thompson, "Clive Thompson on Coding for the Masses", *http://wrd.cm/1uT25O5*

- The `Texting` component for sending texts and processing received texts.
- An input form for submitting the custom response message.
- The `TinyDB` database component for saving the customized message even after the app is closed.
- The `Screen.Initialize` event for loading the custom response when the app launches.
- The `TextToSpeech` component for speaking texts aloud.
- The `LocationSensor` component for reporting the driver's current location.

Getting Started

Open your brower to the App Inventor website and start a new project. Name it "NoTextingWhileDriving" (remember, project names can't have spaces) and set the screen's title to "No Texting While Driving". Then, click Connect and set up live testing on your device or the emulator.

Designing the Components

The user interface for the app is relatively simple: it has a label instructing the user how the app works, a label that displays the text that is to be automatically sent in response to incoming texts, a text box for changing the response, and a button for submitting the change. You'll also need to drag in a `Texting` component, a `TinyDB` component, a `TextToSpeech` component, and a `LocationSensor` component, all of which will appear in the "Non-visible components" area. You can see how this should look in the snapshot of the Component Designer in *Figure 4-2*.

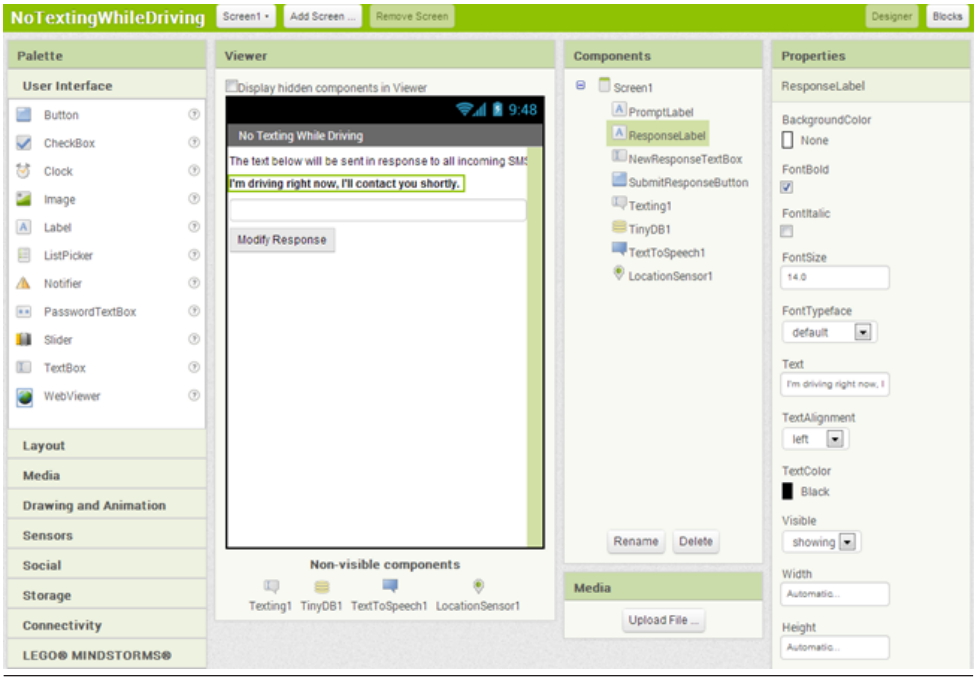

Figure 4-2. The No Texting While Driving app in the Component Designer

You can build the user interface shown in *Figure 4-2* by dragging out the components listed in *Table 4-1*.

Table 4-1. All of the components for the No Texting app

Component type	Palette group	What you'll name it	Purpose
`Label`	User Interface	`PromptLabel`	Let the user know how the app works.
`Label`	User Interface	`ResponseLabel`	The response that will be sent back to the sender.
`TextBox`	User Interface	`newResponseTextBox`	The user will enter the custom response here.
`Button`	User Interface	`SubmitResponseButton`	The user clicks this to submit response.
`Texting`	Social	`Texting1`	Process the texts.
`TinyDB`	Storage	`TinyDB1`	Store the response in the database.
`TextToSpeech`	Media	`TextToSpeech1`	Speak the text aloud.
`LocationSensor`	Sensors	`LocationSensor1`	Sense where the device is.

Set the properties of the components in the following way:

- Set the `Text` of `PromptLabel` to "The text below will be sent in response to all SMS texts received while this app is running."
- Set the `Text` of `ResponseLabel` to "I'm driving right now, I'll contact you shortly." Check its `boldness` property.
- Set the `Text` of `NewResponseTextbox` to " ". (This leaves the text box blank for the user's input.)
- Set the `Hint` of `NewResponseTextbox` to "Enter new response text".
- Set the `Text` of `SubmitResponseButton` to "Modify Response".

Adding Behaviors to the Components

You'll start by programming the autoresponse behavior in which a text reply is sent to any incoming text. You'll then add blocks so that the user can specify a custom response and save that response persistently. Finally, you'll add blocks that read the incoming texts aloud and add location information to the auto-response texts.

AUTO-RESPONDING TO A TEXT

For the auto-response behavior, you'll use App Inventor's `Texting` component. You can think of this component as a little person inside your phone who knows how to read and write texts. For reading texts, the component provides a `Texting.MessageReceived` event block. You can drag this block out and place blocks inside it to show what should happen when a text is received. In the case of this app, we want to automatically send back a text in response.

You can send a text with three blocks. First, you set the phone number to which the text should be sent, which is a property of the `Texting1` component. Next, you set the message to be sent, also a property of `Texting1`. Finally, you actually send the text with the `Texting1.SendMessage` block. *Table 4-2* lists all the blocks you'll need for this auto-response behavior, and *Figure 4-3* shows how they should look in the Blocks Editor.

Table 4-2. The blocks for sending an auto-response

Block type	Drawer	Purpose
`Texting1.MessageReceived`	Texting	The event handler that is triggered when the phone receives a text.
`set Texting1.PhoneNumber to`	Texting	Set the `PhoneNumber` property before sending.
`value number`	Drag from when block	The phone number of the person who sent the text.

Block type	Drawer	Purpose
`set Texting1.Message to`	Texting	Set the **Message** property before sending.
`ResponseLabel.Text`	ResponseLabel	The message the user has entered.
`Texting1.SendMessage`	Texting	Send the message.

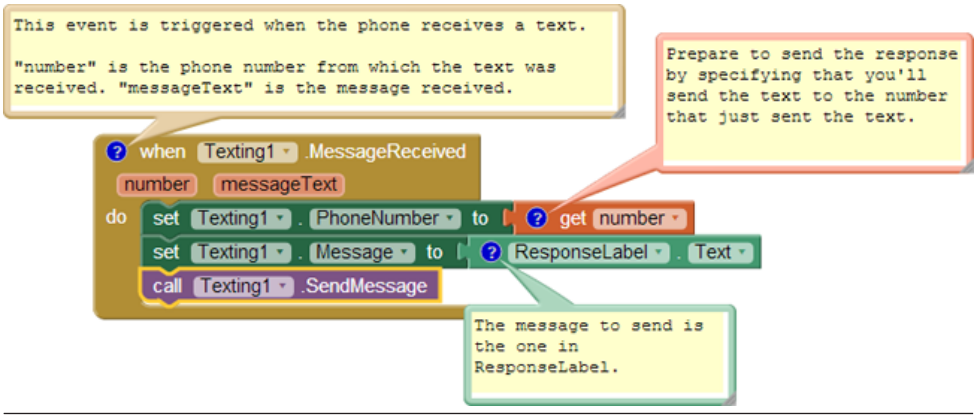

Figure 4-3. Responding to an incoming text

How the blocks work

When the phone receives a text message, the `Texting1.MessageReceived` event is triggered. The phone number of the sender is in the argument `number`, and the message received is in the argument `messageText`.

As the auto-response text should be sent back to the sender, `Texting.PhoneNumber` is set to `number`. `Texting.Message` is set to `ResponseLabel.Text`, which is what you typed while in the Designer: "I'm driving right now, I'll contact you shortly." When these are set, the app calls `Texting.SendMessage` to actually send the response.

Test your app *You'll need two phones to test this behavior, one to run the app and one to send the initial text. If you don't have a second phone handy, you can use Google Voice or a similar service on your computer and send texts from that service to the phone running the app. After you set things up, send a text to the phone running the app. Does the first phone receive the response text?*

ENTERING A CUSTOM RESPONSE

Next, let's add blocks so the user can enter her own custom response. In the Component Designer, you added a `TextBox` component named `NewResponseTextbox`; this is where the user will type the custom response. When the user clicks on the `SubmitResponseButton`, you need to copy the entry (`NewResponseTextbox`) into the `ResponseLabel`, which is used to respond to texts. *Table 4-3* lists the blocks you'll need for transferring a newly entered response into the `ResponseLabel`.

Table 4-3. Blocks for displaying the custom response

Block type	Drawer	Purpose
`SubmitResponseButton.Click`	SubmitResponseButton	The user clicks this button to submit a new response message.
`set ResponseLabel.Text to`	ResponseLabel	Move (set) the newly input value to this label.
`NewResponseTextbox.Text`	NewResponseTextbox	The user has entered the new response here.
`set NewResponseTextbox.Text to`	NewResponseTextbox	Blank out the text box after transferring information
`text` (“”)	Text	The empty text.

How the blocks work

Think of how you interact with a typical input form: you first type something in a text box and then click a submit button to signal the system to process it. The input form for this app is no different. *Figure 4-4* shows how the blocks are programmed so that when the user clicks the `SubmitResponseButton`, the `SubmitResponseButton.Click` event is triggered.

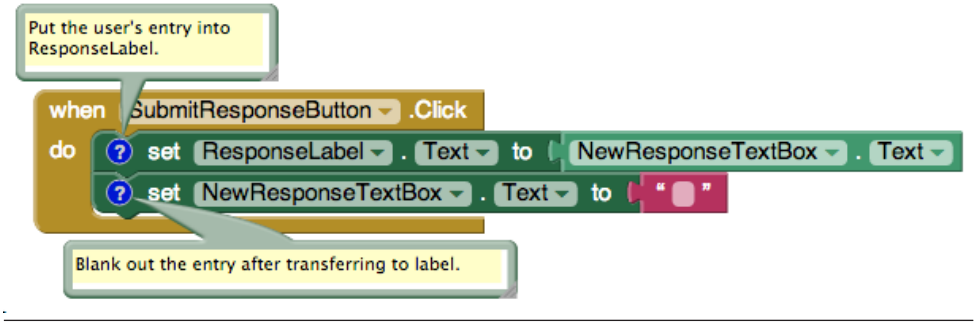

Figure 4-4. Setting the response to the user's entry

The event handler in this case copies (or, in programming terms, *sets*) what the user has entered in `NewResponseTextbox` into the `ResponseLabel`. Recall that `ResponseLabel` holds the message that will be sent out in the auto-response, so you want to be sure to place the newly entered custom message there.

Test your app *Enter a custom response and submit it, and then use the second phone to send another text to the phone running the app. Was the custom response sent?*

STORING THE CUSTOM RESPONSE PERSISTENTLY

Your user can now customize the auto-response, but there is one catch: if the user enters a custom response and then closes the app and relaunches it, the custom response will not appear (instead, the default response will). This behavior is not what your users will expect; they'll want to see the custom response they entered when they restart the app. To make this happen, you need to store that custom response *persistently*.

Placing data in the `ResponseLabel.Text` property is technically storing it, but the issue is that data stored in component properties is *transient* data. Transient data is like your short-term memory; the phone "forgets" it as soon as an app closes. If you want your app to remember something *persistently*, you have to transfer it from short-term memory (a component property or variable) to long-term memory (a database or file).

To store data persistently in App Inventor, you use the `TinyDB` component, which stores data in a file on the Android device. `TinyDB` provides two functions: `StoreValue` and `GetValue`. With the former, the app can store information in the device's database, whereas with the latter, the app can retrieve information that has already been stored.

For many apps, you'll use the following scheme:

1. Store data to the database each time the user submits a new value.
2. When the app launches, load the data from the database into a variable or property.

You'll start by modifying the `SubmitResponseButton.Click` event handler so that it stores the data persistently, using the blocks listed in *Table 4-4*.

Table 4-4. Blocks for storing the custom response with TinyDB

Block type	Drawer	Purpose
`TinyDB1.StoreValue`	TinyDB1	Store the custom message in the phone's database.

Block type	Drawer	Purpose
`text` ("responseMessage")	Text	Use this as the tag for the data.
`ResponseLabel.Text`	ResponseLabel	The response message is now here.

How the blocks work

This app uses `TinyDB` to take the text it just put in `ResponseLabel` and store it in the database. As shown in *Figure 4-5*, when you store something in the database, you provide a tag with it; in this case, the tag is "responseMessage." Think of the tag as the name for the data in the database; it uniquely identifies the data you are storing. As you'll see in the next section, you'll use the same tag ("responseMessage") when you load the data back in from the database.

Figure 4-5. Storing the custom response persistently

RETRIEVING THE CUSTOM RESPONSE WHEN THE APP OPENS

The reason for storing the custom response in the database is so that it can be loaded back into the app the next time the user opens it. App Inventor provides a special event block that is triggered when the app opens: `Screen1.Initialize` (if you completed MoleMash in *Chapter 3*, you've seen this before). If you drag this event block out and place blocks in it, those blocks will be executed immediately when the app launches.

For this app, your `Screen1.Initialize` event handler will load the custom response from the database by using the `TinyDB.GetValue` function. The blocks you'll need for this are shown in *Table 4-5*.

Table 4-5. Blocks for loading the data back in when the app is opened

Block type	Drawer	Purpose
`Screen1.Initialize`	Screen1	This is triggered when the app begins.
`TinyDB1.GetValue`	TinyDB1	Get the stored response text from the database.
`text` ("responseMessage")	Text	Plug this into the `tag` socket of `TinyDB.GetValue`, making sure the text is the same as that used in `TinyDB.StoreValue` earlier.
`text` ("I'm driving right now, I'll contact you shortly")	Text	Plug this into the `valueIfTagNotThere` slot of `TinyDB.GetValue`. This is the default message that should be used if the user has not yet stored a custom response.
`set ResponseLabel.Text to`	ResponseLabel	Place the retrieved value in `ResponseLabel`.

How the blocks work

Figure 4-6 shows the blocks. To understand them, you must envision a user opening the app for the first time, entering a custom response, and opening the app subsequent times. The first time the user opens the app, there won't be any custom response in the database to load, so you want to leave the default response in the `ResponseLabel`. On successive launches, you want to load the previously stored custom response from the database and place it in the `ResponseLabel`.

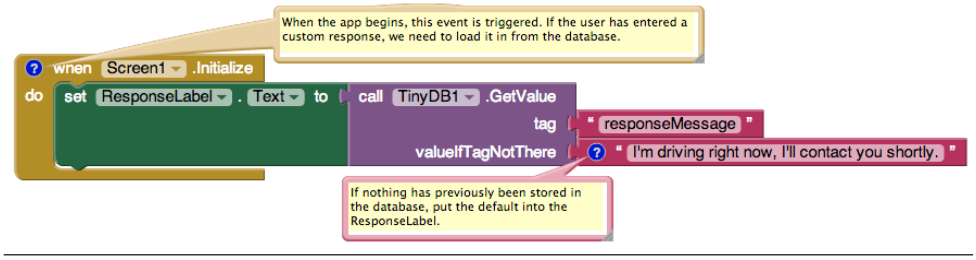

Figure 4-6. Loading the custom response from the database upon app initialization

When the app begins, the `Screen1.Initialize` event is triggered. The app calls the `TinyDB1.GetValue` with a tag of `responseMessage`, the same tag you used when you stored the user's custom response entry earlier. If there is data in the `TinyDB` with a tag of `responseMessage`, it is returned and placed in the `ResponseLabel`.

However, there won't be data the first time the app is launched; this will be the case until the user types a custom response. To handle such cases, `TinyDB1.GetValue` has a second parameter, `valueIfTagNotThere`. If no data is found, the value in `valueIfTagNotThere` is used instead. In this case, "I'm driving right now, I'll contact you shortly," the default value, is placed into `ResponseLabel`.

Test your app To test this behavior, you need to restart your app to see if the data is truly stored persistently and retrieved correctly. In live testing, you can restart the app by changing some component property in the designer, such as the font size of a Label. This will cause the app to reload and `Screen.Initialize` to be triggered. Of course, you can also test the app by actually building it and installing the .apk file on your phone. Once the app is on your phone, launch it, type a message for the custom response, close the app, and then reopen it. If the message you entered is still there, things are working correctly.

SPEAKING THE INCOMING TEXTS ALOUD

In this section, you'll modify the app so that when you receive a text, the sender's phone number, along with the message, is spoken aloud. The idea here is that when you're driving and hear a text come in, you might be tempted to check the text even if you know the app is sending an auto-response. With text-to-speech, you can hear the incoming texts and keep your hands on the wheel.

Android devices provide text-to-speech capabilities, and App Inventor provides a component, `TextToSpeech`, that will speak any text you give it. Note that the "Text" in `TextToSpeech` refers to a sequence of letters, digits, and punctuation, not an SMS text.

The `TextToSpeech` component is very simple to use. You just call its Speak function and plug in the text that you want spoken into its *message* slot. For instance, the blocks shown in *Figure 4-7* would speak the words, "Hello World."

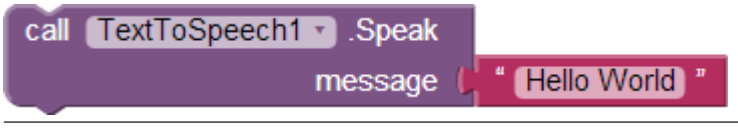

Figure 4-7. Blocks for speaking "Hello World" aloud

For the No Texting While Driving app, you'll need to provide a more complicated message to be spoken, one that includes both the text received and the phone number of the person who sent it. Instead of plugging in a static text object such as the "Hello World" text block, you'll plug in a `join` block. A commonly used function, `join`, makes it possible for you to combine separate pieces of text (or numbers and other characters) into a single text object.

You'll need to make the call to `TextToSpeech.Speak` within the `Texting.MessageReceived` event handler you programmed earlier. The blocks you programmed previously handle this event by setting the `PhoneNumber` and `Message`

properties of the `Texting` component appropriately and then sending the response text. You'll extend that event handler by adding the blocks listed in *Table 4-6*.

Table 4-6. Blocks for speaking the incoming text aloud

Block type	Drawer	Purpose
`TextToSpeech1.Speak`	TextToSpeech1	Speak the message received aloud.
`join`	Text	Concatenate (join together) the words that will be spoken.
`text` ("SMS text received from")	Text	The first words spoken.
`get number`	Drag in from when block	The number from which the original text was received.
`text` (".The message is")	Text	Put a period in after the phone number and then say, "The message is."
`get messageText`	Drag in from when block	The original message received.

How the blocks work

After the response is sent, the `TextToSpeech1.Speak` function is called, as shown at the bottom of *Figure 4-8*. You can plug any text into the message socket of the `TextToSpeech1.Speak` function. In this case, `join` is used to build the words to be spoken—it *concatenates* (or joins) together the text "SMS text received from" and the phone number from which the message was received (`get number`), plus the text ".The message is," and finally the message received (`get messageText`). So, if the text "hello" was sent from the number "111-2222," the phone would say, "SMS text received from 111-2222. The message is hello."

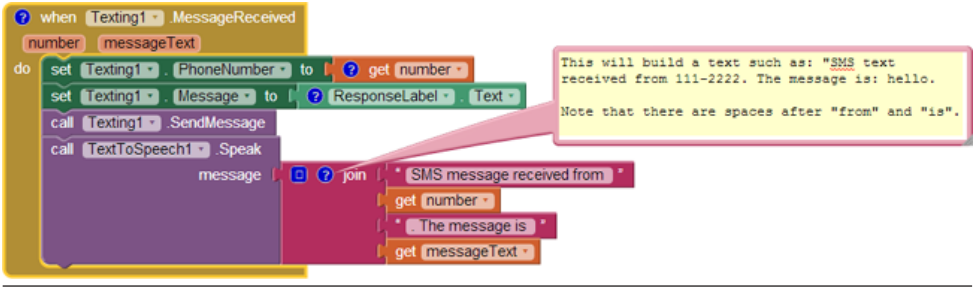

Figure 4-8. Speaking the incoming text aloud

Test your app *You'll need a second phone to test your app. From the second phone, send a text to the phone running the app. Does the phone running the app speak the text aloud? Does it still send an automated response?*

ADDING LOCATION INFORMATION TO THE RESPONSE

Check-In apps help people track one another's location. There are major privacy concerns with such apps, one reason being that location tracking kindles people's fear of a "Big Brother" apparatus that a totalitarian government might set up to track its citizens' whereabouts. But apps that use location information can be quite useful. Think of a lost child, or hikers who've gone off the trail in the woods.

In the No Texting While Driving app, you can use location tracking to convey a bit more information in the auto-response to incoming texts. Instead of just "I'm driving," the response message can be something like, "I'm driving and I'm currently at 3413 Cherry Avenue." For someone awaiting the arrival of a friend or family member, this extra information can be helpful.

App Inventor provides the `LocationSensor` component for interfacing with the phone's GPS (or *Global Positioning System*). Besides latitude and longitude information, the `LocationSensor` can also tap into Google Maps to provide the driver's current street address.

It's important to note that `LocationSensor` doesn't always have a reading. For this reason, you need to take care to use the component properly. Specifically, your app should respond to the `LocationSensor.LocationChanged` event handler. A `LocationChanged` event occurs when the phone's location sensor first gets a reading, and when the phone is moved to generate a new reading. Using the blocks listed in *Table 4-7*, our scheme, shown in *Figure 4-9*, will respond to the `LocationChanged` event by placing the current address in a variable we'll name `lastKnownLocation`. Later, we'll change the response message to incorporate the address we get from this variable.

Table 4-7. Blocks to set up the location sensor

Block type	Drawer	Purpose
`initialize global variable` ("lastKnownLocation")	Variables	Create a variable to hold the last read address.
`text` ("unknown")	Text	Set the default value in case the phone's sensor is not working.
`LocationSensor1.LocationChanged`	LocationSensor1	This is triggered on the first location reading and every location change.
`set global lastKnownLocation to`	Drag from initialize global block.	Set this variable to be used later.
`LocationSensor1.CurrentAddress`	LocationSensor1	This is a street address such as "2222 Willard Street, Atlanta, Georgia."

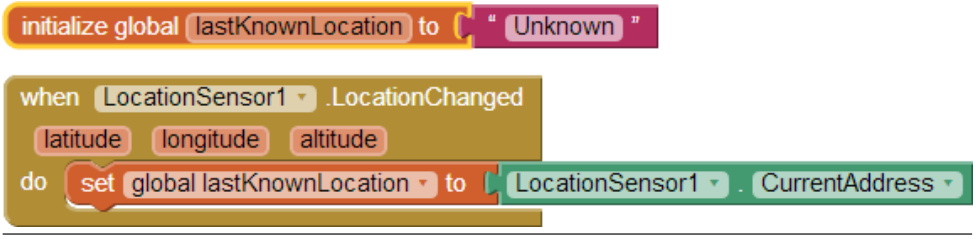

Figure 4-9. Recording the phone's location in a variable each time the GPS location is sensed

How the blocks work

The `LocationSensor1.LocationChanged` event is triggered the first time the sensor gets a location reading and then each time the device is moved so that a new reading is generated. The `LocationSensor1.CurrentAddress` function is called to get the current street address of the device and store it in the `lastKnownLocation` variable.

Note that with these blocks, you've finished only half of the job. The app still needs to incorporate the location information into the auto-response text that will be sent back to the sender. You'll do that next.

SENDING THE LOCATION AS PART OF THE RESPONSE

Using the variable `lastKnownLocation`, you can modify the `Texting1.MessageReceived` event handler to add location information to the response. *Table 4-8* lists the blocks you'll need for this.

Table 4-8. Blocks to display location information in the auto-response

Block type	Drawer	Purpose
`join`	Text	concatenate some text together
`ResponseLabel.Text`	MessageTextBox	This is the (custom) message in the text box.
`text` ("My last known location is:")	Text	This will be spoken after the custom message (note the leading space).
`get global lastKnownLocation`	LocationSensor	This is an address such as "1600 Pennsylvania Ave NW, Washington, DC 20500."

How the blocks work

This behavior works in concert with the `LocationSensor1.LocationChanged` event and the variable `lastKnownLocation`. As you can see in *Figure 4-10*, instead of directly sending a message containing the text in `ResponseLabel.Text`, the app first builds a

message by using `join`. It combines the response text in `ResponseLabel.Text` with the text "My last known location is:" followed by the variable `lastKnownLocation`.

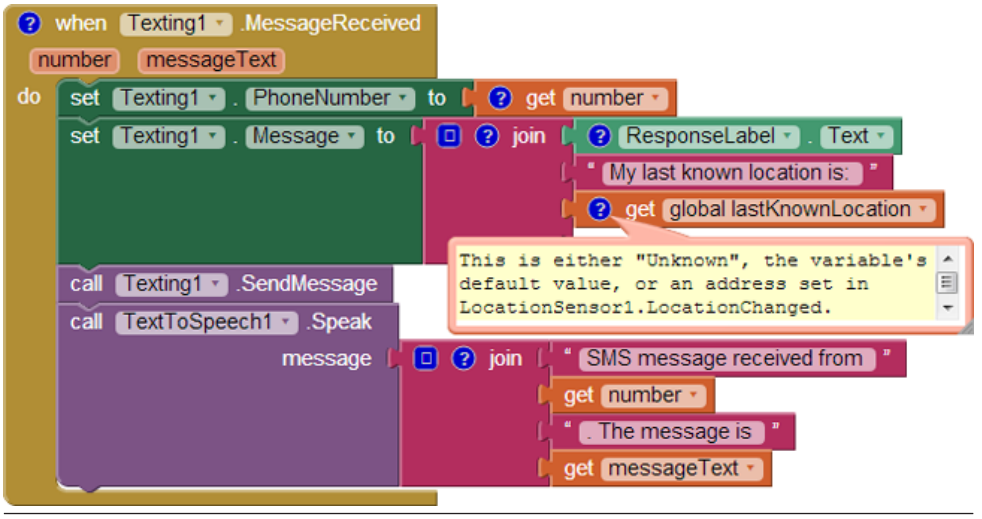

Figure 4-10. Including location information in the response text

The default value of `lastKnownLocation` is "unknown," so if the location sensor hasn't yet generated a reading, the second part of the response message will contain the text, "My last known location is: unknown." If there has been a reading, the second part of the response will be something like, "My last known location is: 1600 Pennsylvania Ave NW, Washington, DC 20500."

Test your app *From the second phone, send a text to the phone running the app. Does the second phone receive the response text with the location information? If it doesn't, make sure you've turned GPS on in the Location settings of the phone running the app.*

The Complete App: No Texting While Driving

Figure 4-11 shows the final block configuration for No Texting While Driving.

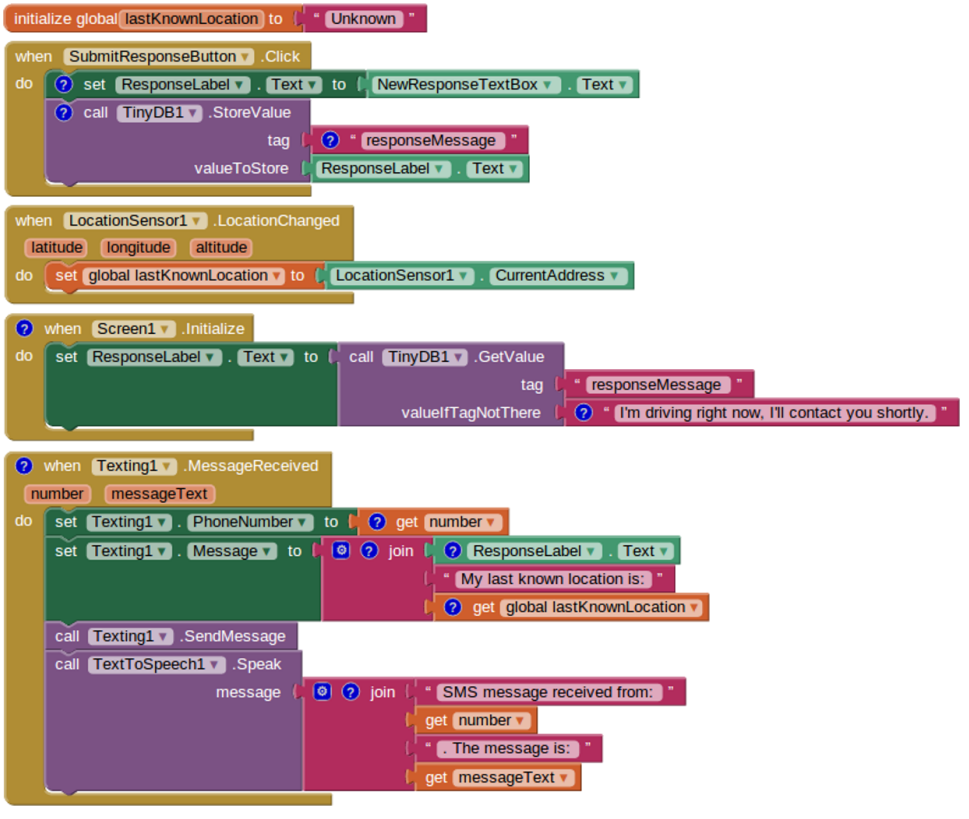

Figure 4-11. The complete No Texting While Driving app

Variations

Once you get the app working, you might want to explore some variations, such as the following:

- Write a version that lets the user define custom responses for particular incoming phone numbers. You'll need to add conditional (`if`) blocks that check for those numbers. For more information on conditional blocks, see *Chapter 18*.
- Write a version that sends custom responses based on whether the user is within certain latitude/longitude boundaries. So, if the app determines that you're in room 222, it will send back "Bob is in room 222 and can't text right now." For more information on the `LocationSensor` and determining boundaries, see *Chapter 23*.
- Write a version that sounds an alarm when a text is received from a number in a "notify" list. For help working with lists, see *Chapter 19*.

Summary

Here are some of the concepts we covered in this tutorial:

- You can use the `Texting` component to both send text messages and process the ones that are received. Before calling `Texting.SendMessage`, you should set the `PhoneNumber` and `Message` properties of the `Texting` component. To respond to an incoming text, program the `Texting.MessageReceived` handler.
- The `TinyDB` component is used to store information persistently—in the phone's database—so that the data can be reloaded each time the app is opened. For more information on `TinyDB`, see *Chapter 22*.
- The `TextToSpeech` component takes any text object and speaks it aloud.
- You can use `join` to piece together (or concatenate) separate text items into a single text object.
- The `LocationSensor` component can report the phone's latitude, longitude, and current street address. To ensure that it has a reading, you should access its data within the `LocationSensor.LocationChanged` event handler, which is triggered the first time a reading is made and upon every change thereafter. For more information on the `LocationSensor`, see *Chapter 23*.

If you're interested in exploring SMS-processing apps further, check out the Broadcast Hub app in *Chapter 11*.