CHAPTER 10

MakeQuiz and TakeQuiz

You can customize the Presidents Quiz app in Chapter 8 to build any quiz, but it is only the programmer who can modify the questions and answers. There is no way for parents, teachers, or other app users to create their own quizzes or change the quiz questions (unless they too want to learn how to use App Inventor!).

In this chapter, you'll build a MakeQuiz app that lets a "teacher" create quizzes using an input form. The quiz questions and answers will be stored in a web database so that "students" can access a separate TakeQuiz app and take the test. While building these two apps, you'll make yet another significant conceptual leap: learning how to create apps with *user-generated* data that is shared across apps and users.

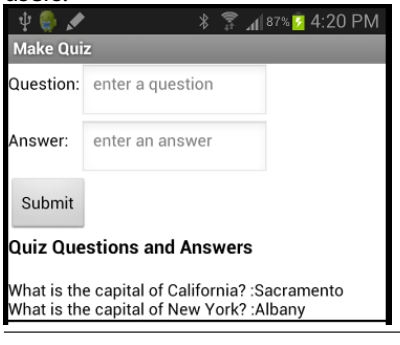

Figure 10-2. The MakeQuiz app in action

Parents can create fun trivia apps for their children during a long road trip, grade-school teachers can build "Math Blaster" quizzes, and college students can build quizzes to help their study groups prepare for a final. This chapter builds on the Presidents Quiz in *Chapter 8*, so if you haven't completed that app, you should do so before continuing on here.

You'll design two apps, MakeQuiz for the teacher (see *Figure 10-1*) and TakeQuiz for the student, which will appear similar to the Presidents Quiz.

Figure 10-1.

Here are the behaviors you'll code for the first app, MakeQuiz:

- The user types questions and answers in an input form.
- The question-answer pairs are displayed.
- The quiz questions and answers are stored in a web database.

The second app you'll create, TakeQuiz, will work similarly to the Presidents Quiz app you've already built. In fact, you'll use the Presidents Quiz app as a starting point. TakeQuiz will differ in that the questions asked will be those that were entered into the database via MakeQuiz.

What You'll Learn

The Presidents Quiz was an example of an app with static data: no matter how many times you take the quiz, the questions are always the same because they are *hardcode*d into the app; that is, the questions and answers are part of the blocks. News apps, blogs, and social networking apps such as Facebook and Twitter work with *dynamic* data, meaning the data can change over time. Often, this dynamic information is user generated—the app allows users to enter, modify, and share information. With MakeQuiz and TakeQuiz, you'll learn how to build an app that handles shared, user-generated data.

If you completed the Xylophone app (*Chapter 9*), you've already been introduced to dynamic lists; in that app, the musical notes the user plays are recorded in lists. Apps with such user-generated data are more complex, and the blocks are more abstract because they don't rely on predefined, static data. You define list variables, but you define them without specific items. As you program your app, you need to envision the lists being populated with data provided by the end user.

This tutorial covers the following App Inventor concepts:

- Input forms for allowing the user to enter information.
- Using an indexed list along with `for each` to display items from multiple lists.
- Persistent list data—MakeQuiz will save the quiz questions and answers in a web database, and TakeQuiz will load them in from the same database.
- Data sharing—you'll store the data in a web database by using the `TinyWebDB` component (instead of the `TinyDB` component used in previous chapters).

Getting Started

Connect to the App Inventor website and start a new project. Name it "MakeQuiz" and set the screen's title to "Make Quiz". Connect your app to your device or emulator for live testing.

Designing the Components

Use the Component Designer to create the interface for MakeQuiz. When you finish, it should look something like *Figure 10-2* (there are also more detailed instructions after the snapshot).

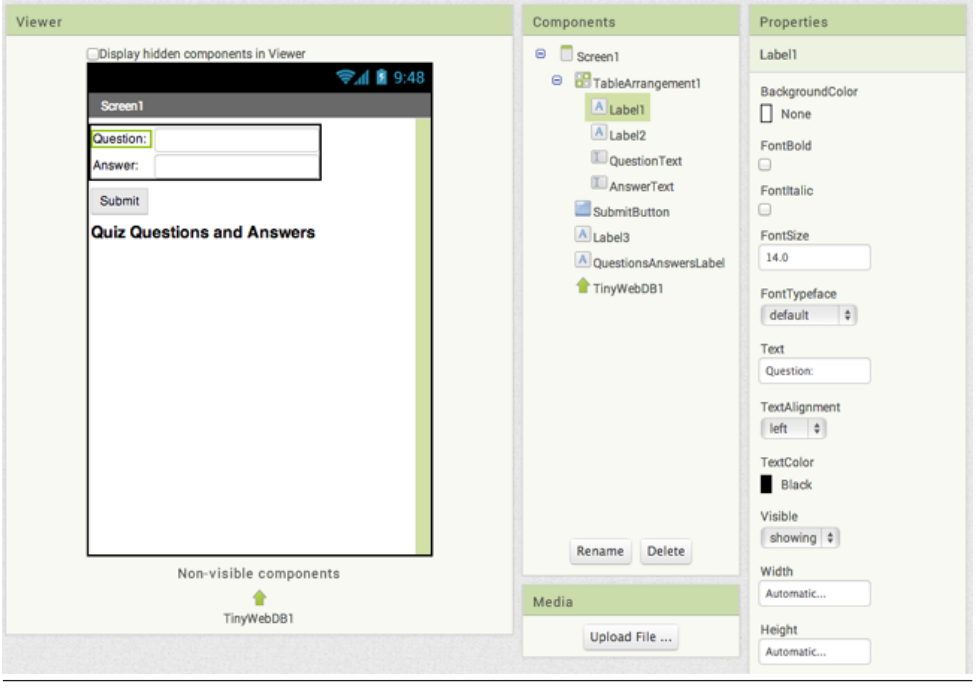

Figure 10-3. MakeQuiz in the Component Designer

You can build the user interface shown in *Figure 10-2* by dragging out the components listed in *Table 10-1*. Drag each component from the Palette into the Viewer and name it as specified in the table. Note that you can leave the header label names (Label1 – Label4) as their defaults (you won't use them in the Blocks Editor anyway).

Table 10-1. All the components for the MakeQuiz app

Component type	Palette group	What you'll name it	Purpose
`TableArrangement`	Layout	`TableArrangement1`	Format the form, including the question and answer.
`Label`	User Interface	`Label1`	The "Question:" prompt.
`TextBox`	User Interface	`QuestionText`	The user enters questions here.

Component type	Palette group	What you'll name it	Purpose
`Label`	User Interface	`Label2`	The "Answer:" prompt.
`TextBox`	User Interface	`AnswerText`	The user enters answers here.
`Button`	User Interface	`SubmitButton`	The user clicks this to submit a QA pair.
`Label`	User Interface	`Label3`	Display "Quiz Questions and Answers."
`Label`	User Interface	`QuestionsAnswersLabel`	Display previously entered QA pairs.
`TinyWebDB`	Storage	`TinyWebDB1`	Web storage for QA pairs.

Set the properties of the components in the following way:

1. Set the Text of `Label1` to "Question", the Text of `Label2` to "Answer", and the text of `Label3` to "Quiz Questions and Answers".
2. Set the FontSize of `Label3` to 18 and check the FontBold box.
3. Set the Hint of `QuestionText` to "Enter a question" and the Hint of AnswerText to "Enter an answer".
4. Set the Text of `SubmitButton` to "Submit".
5. Set the Text of `QuestionsAnswersLabel` to "Quiz Questions and Answers".
6. Move the `QuestionText`, `AnswerText`, and their associated labels into `TableArrangement1`.

If you look at the properties for `TinyWebDB`, you'll notice that it has a property `ServiceURL` (see *Figure 10-3*). This property specifies a web database service, specially configured to work with the `TinyWebDB` component, where your shared data will be stored. By default, the web service it refers to is one set up by the MIT App Inventor team at *http://appinvtinywebdb.appspot.com*. You'll use this default service in this tutorial as you work; however, it is important to know that anyone using App Inventor will be storing information to this same web service, and that the data your app puts there will be seen by all, and might even be overwritten by someone.

The default service is for testing only. It is fairly easy (and free!) to configure your own such service, which you'll want to do if you build an app that will be deployed with real users. For now, continue on and complete this tutorial, but when you're ready the instructions for setting up your own web service are at *"TinyWebDB and TinyWebDB-Compliant APIs" on page 368*.

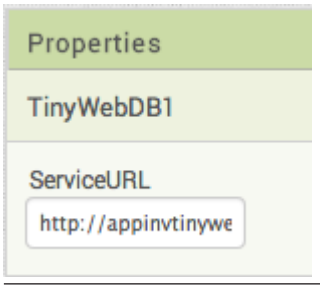

Figure 10-4. With TinyWebDB.ServiceURL, you can specify the URL of a web database you set up

Adding Behaviors to the Components

As with the Presidents Quiz app, you'll first define some global variables for the QuestionList and AnswerList, but this time you won't provide fixed questions and answers.

CREATING EMPTY QUESTION AND ANSWER LISTS

The blocks for the lists should look as shown in *Figure 10-4*.

Figure 10-5. The lists for MakeQuiz begin empty

The lists are defined with the `create empty list` block, instead of the `make a list` block. This is because with the MakeQuiz and TakeQuiz apps, all data will be created by the app user (it is dynamic, user-generated data).

RECORDING THE USER'S ENTRIES

The first behavior you'll build is for handling the user's input. Specifically, when the user enters a question and answer and clicks Submit, you'll use `add items to list` blocks to update the `QuestionList` and `AnswerList`. The blocks should appear as illustrated in *Figure 10-5*.

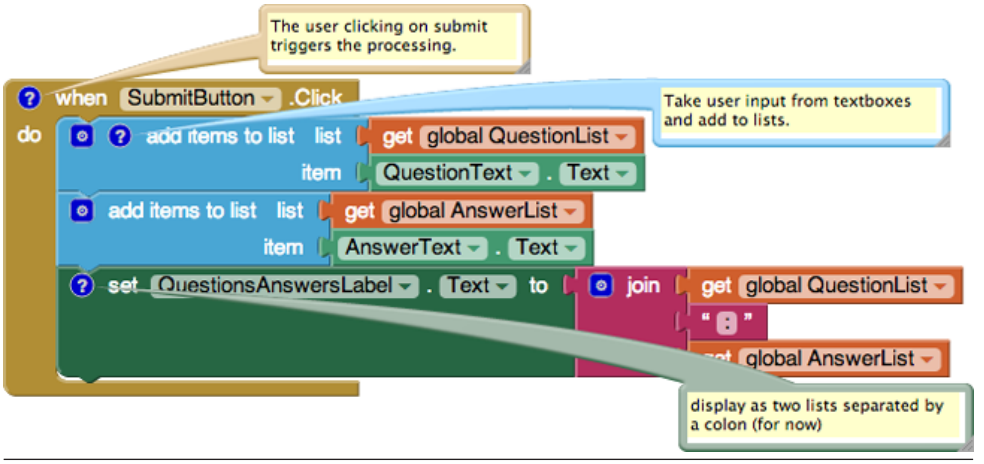

Figure 10-6. Adding new entries to the lists

How the blocks work

The `add items to list` block appends each item to the end of a list. As shown in *Figure 10-5*, the app takes the text the user has entered in the `QuestionText` and `AnswerText` text boxes and appends each to the corresponding list.

The `add items to list` blocks update the `QuestionList` and `AnswerList` variables, but these changes are not yet shown to the user. The third row of blocks displays these lists by concatenating them (joining them) with a colon inserted between. By default, App Inventor displays lists with surrounding parentheses and spaces between items: for example, "(item1 item2 item3)." Of course, this is not the ideal way to display the lists, but it will allow you to test the app's behavior for now. Later, you'll create a more sophisticated method of displaying the lists that shows each question-answer pair on a separate line.

BLANKING OUT THE QUESTION AND ANSWER

Recall from the Presidents Quiz app that when you moved on to the next question in the list, you needed to blank out the user's answer from the previous question. In this app, when a user submits a question-answer pair, you'll want to clear the `QuestionText` and `AnswerText` text boxes so that they're ready for a new entry instead of showing the previous one. The blocks should appear as those shown in *Figure 10-6*.

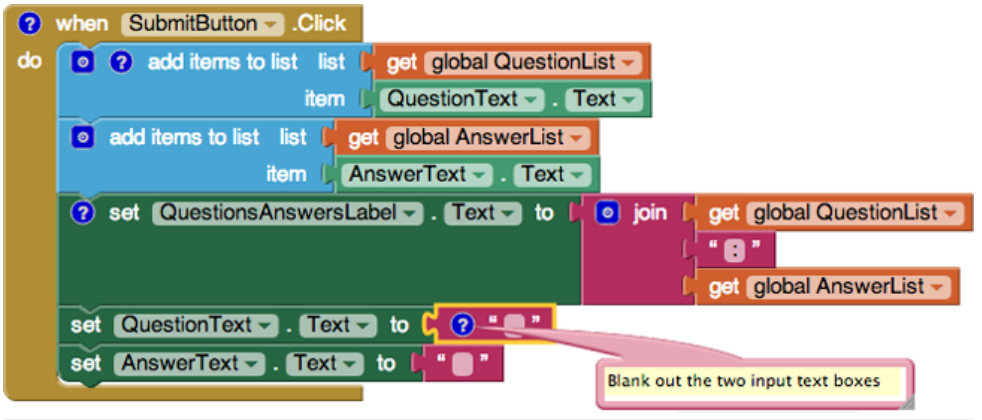

Figure 10-7. Blanking out the question and answer text boxes after submission

Test your app *Test the behavior by entering a couple of question-answer pairs. As you add them, do they appear below the form in the* `QuestionsAnswersLabel`*?*

How the blocks work

When the user submits a new question and answer, they are added to their respective lists and displayed. At that point, the text in the `QuestionText` and `AnswerText` is blanked out with empty text blocks.

DISPLAYING QUESTION-ANSWER PAIRS ON MULTIPLE LINES

In the app you've built so far, the question and answer lists are displayed separately and using the default list display format for App Inventor. So if you were making a quiz on state capitals and had entered two pairs of questions and answers, it might appear as:

(What is the capital of California? What is the capital of New York?: Sacramento Albany)

This is obviously not an ideal user interface for the quiz designer. A better display would show each question along with its corresponding answer, with one question-answer pair per line, like this:

What is the capital of California?: Sacramento
What is the capital of New York?: Albany

The technique for displaying a single list with each item on a separate line is described in *Chapter 20*—you might want to read that chapter before going on.

The task here is a bit more complicated because you're dealing with two lists. Because of its complexity, you'll put the blocks for displaying the data in a procedure

named `displayQAs`, and call that procedure from the `SubmitButton.Click` event handler.

To display question-answer pairs on separate lines, you'll need to do the following:

- Use a `for each` block to iterate through each question in the `QuestionList`.
- Use a variable `answerIndex` so that you can grab each answer as you iterate through the questions.
- Use `join` to build a text object with each question and answer pair, and a newline character (\n) separating each pair.

The blocks should appear as illustrated in *Figure 10-7*.

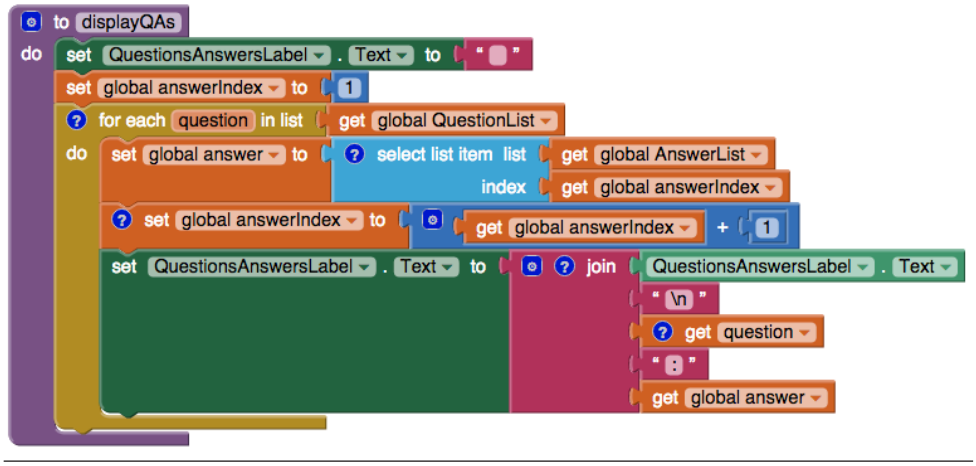

Figure 10-8. The displayQAs procedure

How the blocks work

The `displayQAs` procedure encapsulates all of the blocks for displaying the data.

By using a procedure, you won't have to copy the blocks needed to display the list more than once in the app—you can just call `displayQAs` when you need to display the lists.

The `for each` only allows you to iterate through a single list. In this case, there are two lists, `QuestionList` and `AnswerList`. The `for each` is used to iterate through the `QuestionList`, but you need to select an answer, as well, as you proceed through the questions. To accomplish this, you use an index variable, as was done with the `currentQuestionIndex` in the Presidents Quiz tutorial in *Chapter 8*. In this case, the index variable, `answerIndex`, is used to track the position in the `AnswerList` as the `for each` goes through the `QuestionList`.

`answerIndex` is set to 1 before the `for each` begins. Within the `for each`, `answerIndex` is used to select the current answer from the `AnswerList`, and then it is incremented. On each iteration of the `for each`, the current question and answer are concatenated to the end of the `QuestionsAnswersLabel.Text` property, with a colon between them.

CALLING THE DISPLAYQAS PROCEDURE

You now have a procedure for displaying the question-answer pairs, but it won't help unless you call it when you need it. Modify the `SubmitButton.Click` event handler by calling `displayQAs` instead of displaying the lists, as was done previously. The updated blocks should appear as shown in *Figure 10-8*.

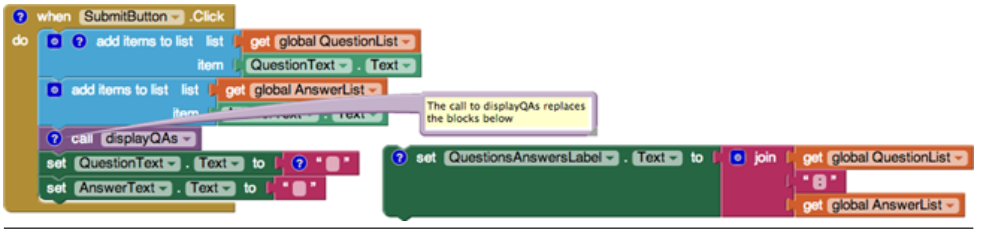

Figure 10-9. Calling the displayQAs procedure to replace the blocks shown to the right

Test your app *Test the behavior by entering a couple of question-answer pairs. As you add them, do they appear on separate lines in the* `QuestionsAnswersLabel`?

SAVING THE QAS PERSISTENTLY ON THE WEB

So far, you've created an app that places the entered questions and answers into a list. But what happens if the quiz maker closes the app? If you've completed the No Texting While Driving app (*Chapter 4*) or the Android, Where's My Car? app (*Chapter 7*), you know that if you don't store the data in a database, it won't be there when the user exits and restarts the app. Storing the data persistently will allow the quiz maker to view or edit the latest update of the quiz each time the app is started. Persistent storage is also necessary because the TakeQuiz app needs access to the data, as well.

You're already familiar with using the `TinyDB` component to store and retrieve data in a database. But in this case, you'll use the `TinyWebDB` component, instead. Whereas TinyDB stores information directly on a phone, `TinyWebDB` stores data in databases that reside on the Web.

What about your app design would merit using an online database instead of one stored on a person's phone? The key issue here is that you're building two apps that

both need access to the same data—if the quiz maker stores the questions and answers on her phone, the quiz takers won't have any way of getting to the data for their quiz! Because `TinyWebDB` stores data on the Web, the quiz taker can access the quiz questions and answers on a different device than the quiz maker's. (Online data storage is often referred to as *the cloud*.)

Here's the general scheme for making list data (such as the questions and answers for our app) persistent:

- Store a list to the database each time a new item is added to it.
- When the app launches, load the list from the database into a variable.

Start by storing the `QuestionList` and `AnswerList` in the database each time the user enters a new pair.

How the blocks work

The `TinyWebDB1.StoreValue` blocks store data in a web database. `StoreValue` has two arguments: the tag that identifies the data, and the value that is the actual data you want to store. *Figure 10-9* shows that the `QuestionList` is stored with a tag of "questions," whereas the `AnswerList` is stored with a tag of "answers."

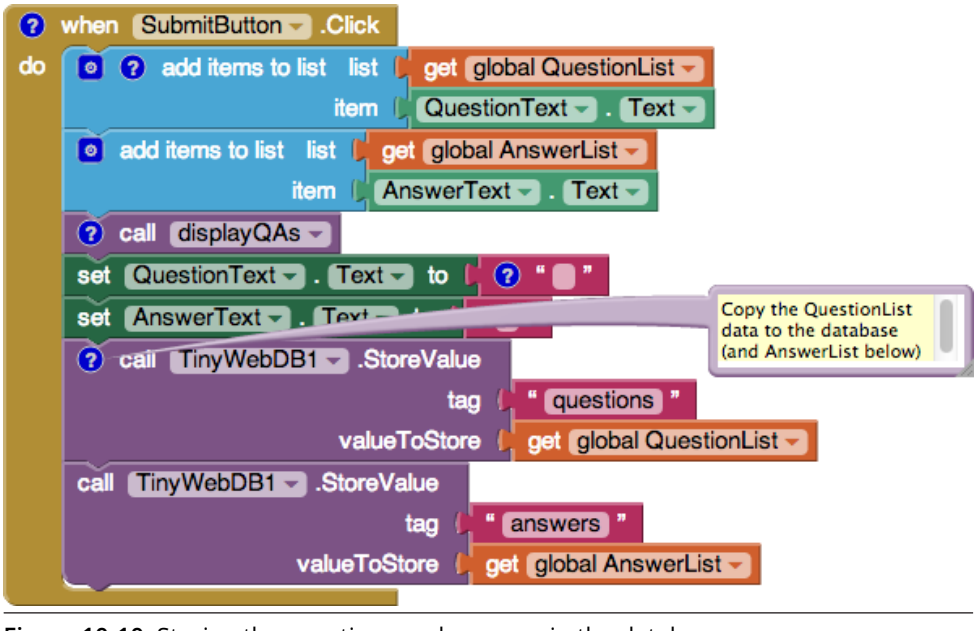

Figure 10-10. Storing the questions and answers in the database

For your app, you should use tags that are more distinctive than "questions" and "answers" (e.g., "DavesQuestions" and "DavesAnswers"). This is important because, at

least initially, you're using the default web database service for App Inventor, which means that others can overwrite your questions and answers, including other people following this tutorial.

Test your app *Testing this part of the app is different from tests you've performed previously because your app is now affecting another entity, the default* `TinyDBWeb` *service. Run the app, enter a question and answer, and then open a browser window to the default web service at http://appinvtinywebdb.appspot.com. Then click "get_value" and enter one of your tags (in this sample, "questions" or "answers"). If things are working correctly, your question and answer lists should appear.*

As mentioned earlier, the default web service is shared among programmers and apps, so it is intended only for testing. When you're ready to deploy your app with real users, you'll want to set up your own private database service. Fortunately, doing so is straightforward and requires no programming (see *"TinyWebDB and TinyWebDB-Compliant APIs" on page 368*).

LOADING DATA FROM THE DATABASE

One reason we need to store the questions and answers in a database is to make it possible for the person creating the quiz to close the app and relaunch it at a later time without losing the questions and answers previously typed. (We also do it so that the quiz taker can access the questions, but we'll cover that later.) Let's program the blocks for loading the lists back into the app from the web database each time the app is restarted.

As we've covered in earlier chapters, to specify what should happen when an app launches, you program the `Screen.Initialize` event handler. In this case, the app needs to request two lists from the `TinyWebDB` web database—the questions and the answers—so the Screen1.Initialize will make two calls to `TinyWebDB.GetValue`. The blocks should appear as depicted in *Figure 10-10*.

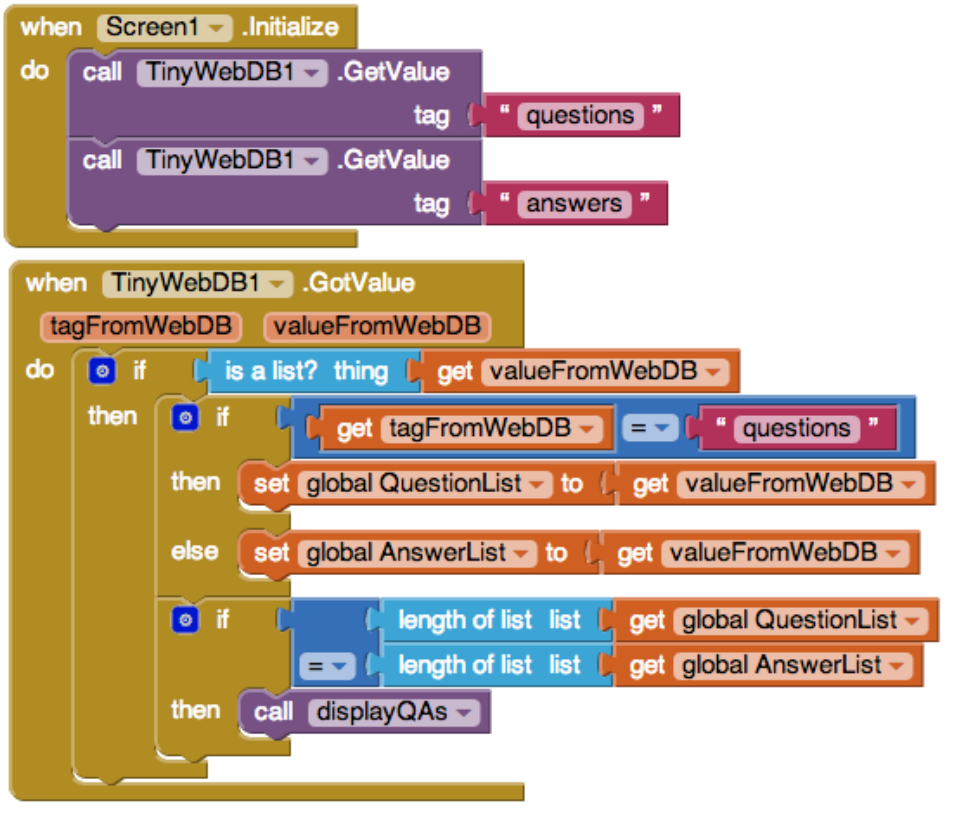

Figure 10-11. Requesting the lists from the database when the app opens and processing when lists arrive

How the blocks work

The `TinyWebDB.GetValue` blocks in *Figure 10-10* work differently than `TinyDB.GetValue`, which returns a value immediately. `TinyWebDB.GetValue` only requests the data from the web database; it doesn't immediately receive a value. Instead, when the data arrives from the web database, a `TinyWebDB.GotValue` event is triggered. You must also program that event handler to process the data that is returned.

When the `TinyWebDB.GotValue` event occurs, the data requested is contained in an argument named `valueFromWebDB`. The tag you requested is contained in the argument `tagFromWebDB`.

In this app, because two different requests are made for the questions and answers, `GotValue` will be triggered twice. To avoid putting questions in your `AnswerList`, or vice versa, your app needs to check the tag to see which request has arrived and then put the value returned from the database into the corresponding list (`QuestionList` or `AnswerList`).

In `Screen.Initialize`, the app calls `TinyWebDB1.GetValue` twice: once to request the stored `QuestionList`, and once to request the stored `AnswerList`. When the data arrives from the web database from either request, the `TinyWebDB1.GotValue` event is triggered.

The `valueFromWebDB` argument of `GotValue` holds the data returned from the database request. You need the outer `if` block in the event handler because the database will return an empty text ("") in `valueFromWebDB` if it's the first time the app has been used and there aren't yet questions and answers. By asking if the `valueFromWebDB is a list?`, you're making sure there is some data actually returned. If there isn't any data, you'll bypass the blocks for processing it.

If data is returned (`is a list?` is true), the blocks go on to check which request has arrived. The tag identifying the data is in `tagFromWebDB`: it will be either "questions" or "answers." If the tag is "questions," the `valueFromWebDB` is put into the variable `QuestionList`. Otherwise (else), it is placed in the `AnswerList`. (If you used tags other than "questions" and "answers," check for those, instead.)

You only want to display the lists after both have arrived (`GotValue` has been triggered twice). Can you think of how you'd know for sure that you have both lists loaded in from the database? The blocks shown use an `if` test to check whether the lengths of the lists are the same, as this can only be true if both lists have been returned. If they are, the handy `displayQAs` procedure you wrote earlier is called to display the loaded data.

The Complete App: MakeQuiz

Figure 10-11 shows the blocks for the entire MakeQuiz app.

TakeQuiz: An App for Taking the Quiz in the Database

You now have a MakeQuiz app that will store a quiz in a web database. Building TakeQuiz, the app that dynamically loads the quiz, is simpler. You can build it with a few modifications to the Presidents Quiz you completed in *Chapter 8* (if you have not completed that tutorial, do so now before continuing).

Begin by opening your Presidents Quiz app in App Inventor, choosing Save As, and naming the new project "TakeQuiz". This will leave your Presidents Quiz app unmodified but now you can use its blocks as the basis for TakeQuiz.

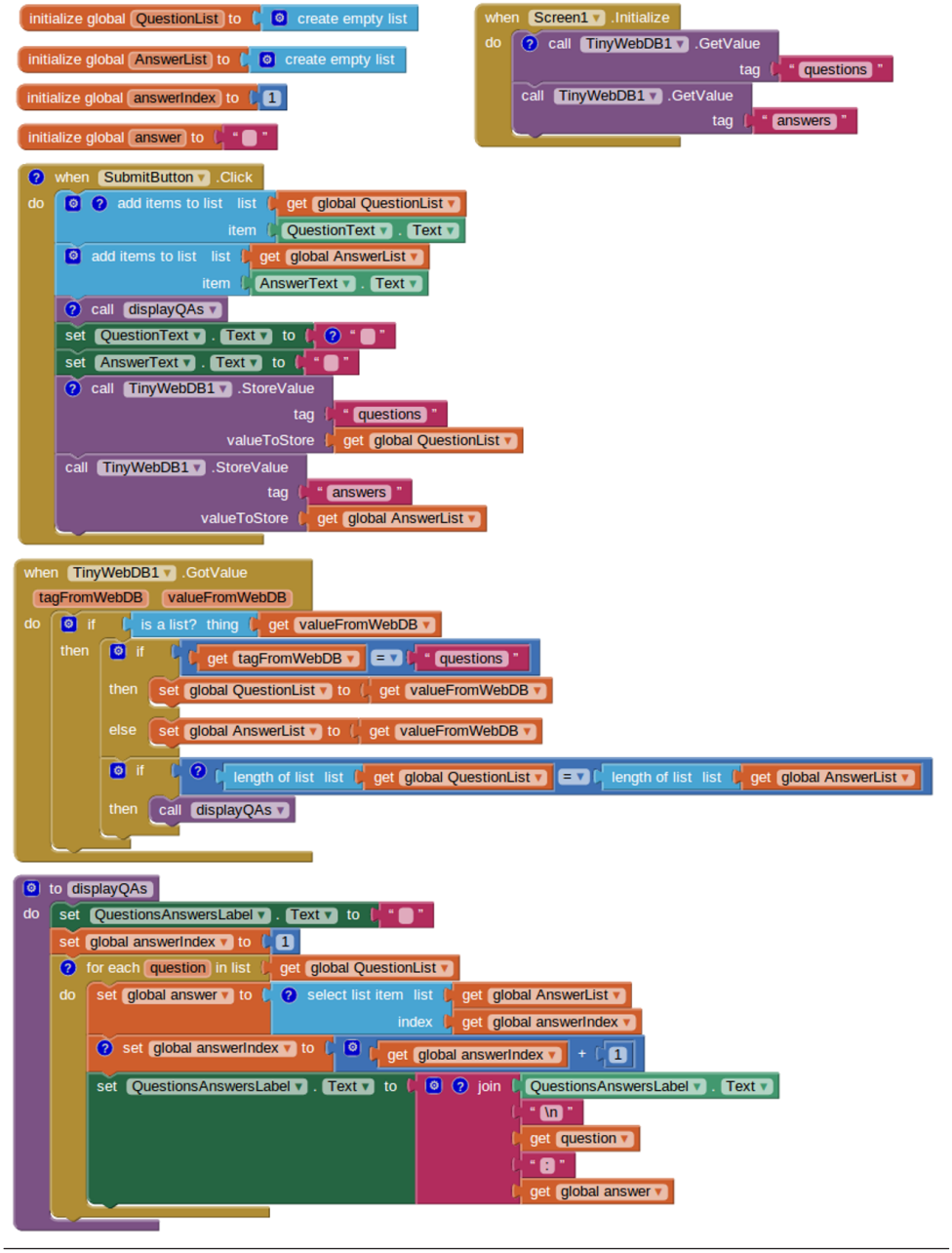

Figure 10-12. The blocks for MakeQuiz

Next, make the following changes in the Designer:

1. This version will not display images with each question, so first remove the references to images from the TakeQuiz app. In the Component Designer,

choose each image from the Media palette and delete it. Then, delete the `Image1` component, which will remove all references to it from the Blocks Editor.

2. Because TakeQuiz will work with database data that resides on the Web, drag a `TinyWebDB` component into the app.
3. Because you don't want the user to answer or click the `NextButton` until the questions are loaded, uncheck the Enabled property of the `AnswerButton` and `NextButton`.

Now, modify the blocks so that the quiz given to the user is loaded from the database. First, because there are no fixed questions and answers, remove all the actual question and answer text blocks from the `make a list` blocks within the `QuestionList` and `AnswerList`. The resulting blocks should appear as shown in *Figure 10-12*.

Figure 10-13. The question and answer lists now start empty

You can also completely delete the `PictureList`; this app won't deal with images. Now, modify your `Screen1.Initialize` so that it calls `TinyWebDB.GetValue` twice to load the lists, just as you did in MakeQuiz. The blocks should look as they do in *Figure 10-13*.

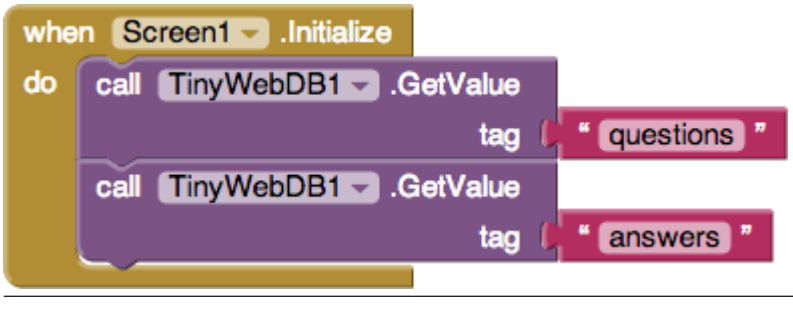

Figure 10-14. Requesting the questions and answers from the web database

Finally, drag out a `TinyWebDB.GotValue` event handler. This event handler should look similar to the one used in MakeQuiz, but here you want to show only the first question and none of the answers. Try making these changes yourself first, and then take a look at the blocks in *Figure 10-14* to see if they match your solution.

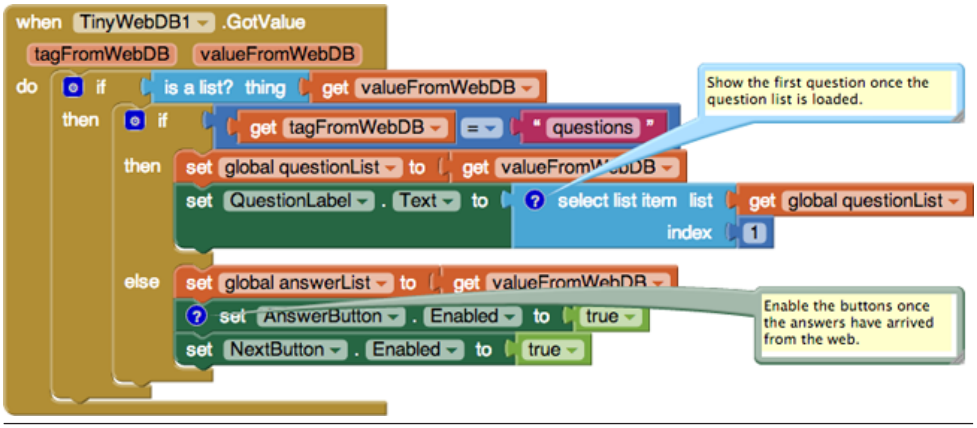

Figure 10-15. GotValue handles the data that arrives from the Web

HOW THE BLOCKS WORK

When the app starts, `Screen1.Initialize` is triggered and the app requests the questions and answers from the web database. When each request arrives, the `TinyWebDB.GotValue` event handler is triggered. The app first checks if there is indeed data in `valueFromWebDB` using `is a list?` If it finds data, the app asks which request has come in, using `tagFromWebDB`, and places the `valueFromWebDB` into the appropriate list. If the `QuestionList` is being loaded, the first question is selected from `QuestionList` and displayed. If the AnswerList is being loaded, the `AnswerButton` and `NextButton` are enabled so that the user can begin taking the test.

These are all the changes you need for TakeQuiz. If you've added some questions and answers with MakeQuiz and you run TakeQuiz, the questions that appear should be the ones you input.

The Complete App: TakeQuiz

Figure 10-15 shows the blocks for the entire TakeQuiz app.

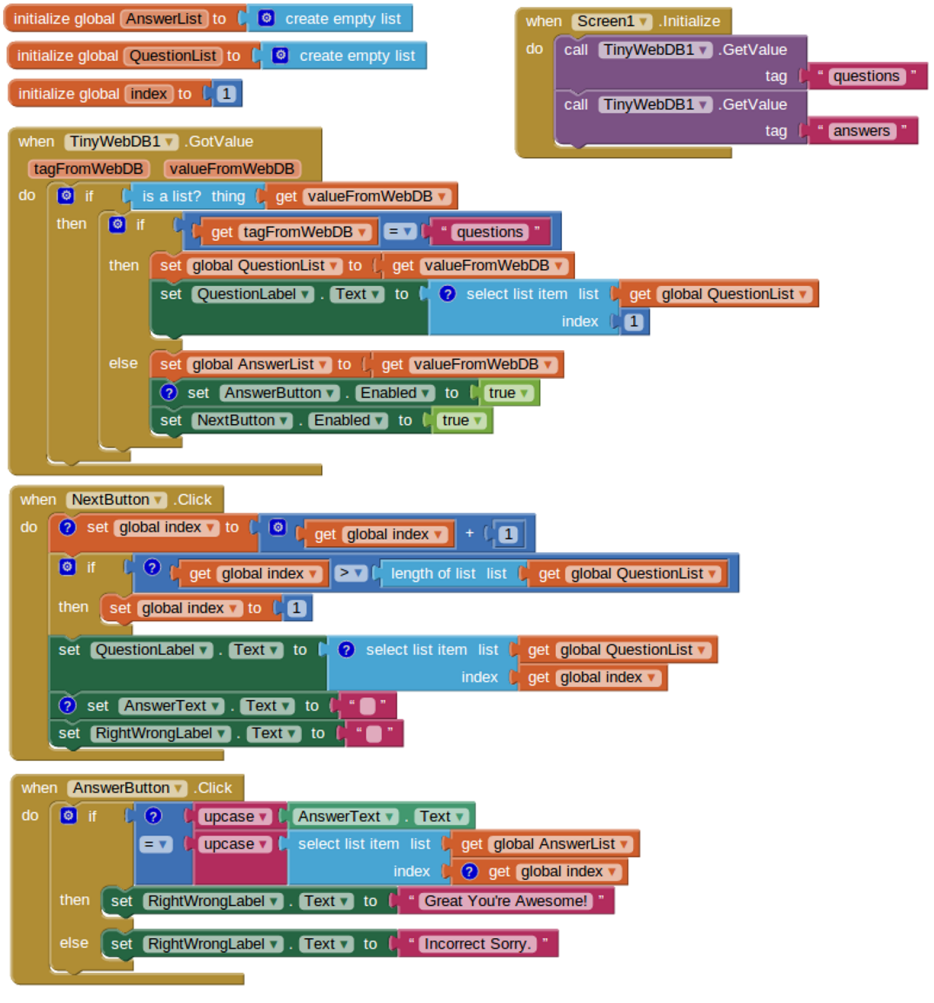

Figure 10-16. The final blocks for TakeQuiz

Variations

After you get MakeQuiz and TakeQuiz working, you might want to explore some of the following variations:

- Allow the quiz maker to specify an image for each question. This is a little complicated because `TinyWebDB` doesn't allow you to store images. Therefore, the images will need to be URLs to pictures on the Web, and the quiz maker will need to enter these URLs as a third item in the MakeQuiz form. Note that you can set the Picture property of an `Image` component to a URL.

- Allow the quiz maker to delete items from the questions and answers. You can let the user choose a question by using the `ListPicker` component, and you can remove an item with the `remove list item` block (remember to remove from both lists and update the database). For help with `ListPicker` and list deletion, see *Chapter 19*.
- Let the quiz maker name the quiz. You'll need to store the quiz name under a different tag in the database, and you'll need to load the name along with the quiz in TakeQuiz. After you've loaded the name, use it to set the `Screen.Title` property so that it appears when the user takes a quiz.
- Allow multiple named quizzes to be created. You'll need a list of quizzes, and you can use each quiz name as (part of) the tag for storing its questions and answers.

Summary

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

- Dynamic data is information input by the app's user or loaded in from a database. A program that works with dynamic data is more abstract. For more information, see *Chapter 19*.
- You can store data persistently in a web database with the `TinyWebDB` component.
- You retrieve data from a `TinyWebDB` database by requesting it with `TinyWebDB.GetValue`. When the web database returns the data, the `TinyWebDB.GotValue` event is triggered. In the `TinyWebDB.GotValue` event handler, you can put the data in a list or process it in some way.
- `TinyWebDB` data can be shared among multiple phones and apps. For more information on (web) databases, see *Chapter 22*.