Using Google App Inventor to Design Mobile Applications
David Wolber, University of San Francisco
November 3, 2010

SESSION 1: 14:30-16:30
Introduction to App Inventor
https://appinventor.org/triesteintro

Getting Started (Demo)
http://appinventor.googlelabs.com/learn/gettingstarted.html
No Text Demo

Tutorial: Hello Purr
http://appinventor.googlelabs.com/learn/tutorials/hellopurr/hellopurr-part1.html

Tutorial: Mole Mash
http://appinventor.googlelabs.com/learn/tutorials/molemash/molemash.html

Discussion: Capabilities and Limitations
http://www.appinventor.org/capabilities-limitations

SESSION 2: 17:00-19:00
Discussion: Teaching Strategies
appinventor.org/app-inventor-and-computer-science-for-non-cs

Tutorial: Quiz Me
http://appinventor.googlelabs.com/learn/tutorials/quizme/quizme.html

Demo: Android, Where’s My Car?
http://appinventor.googlelabs.com/learn/tutorials/whereismycar/whereismycar.html

Challenge: Build Where Are We? -- show location of app users on an image of the university

Play Time: Build Android Where Am I?, a quiz, anything you want.
Sample App: *No Text While Driving*

Components
Hello Purr - A Tutorial Introduction to App Inventor (Part 1)

What You're Building

In this tutorial, you'll start from scratch and use App Inventor to build an Android app on your computer and run it on the phone.

Hello Purr shows a picture of a kitty, and a label that instructs users to pet the kitty. The picture is also a button, and when users tap the button, the phone plays a meow sound.

You'll create the Hello Purr app from scratch. Here's a sneak preview of the steps you'll go through as you work through the tutorial, below.

- Start a new project on the App Inventor site.
- Add components to the app. First, add a label component that reads "Pet the Kitty".
- Next, add a button component with the kitty picture.
- Add a sound component to play the meow sound.
- Create an event handler that tells the sound component to play when the user taps the button.

Getting Started

Before starting work, be sure to set your computer and phone to use App Inventor, by completing the setup procedure. As part of the setup, create a new project and name it HelloPurr, open the Blocks editor, and connect to the phone. Then switch back to the browser window.

At this point, you should have http://appinventor.googlelabs.com in your web browser. Your phone should be connected, with the App Inventor Phone Application running. The browser should be showing the Designer page. Here's what it should look like:

Building apps: Overview

You build App Inventor projects by combining standard components. Components are the basic elements you use to make apps on the Android phone. They're like the ingredients in a recipe. Some components are very simple, like a Label component, which just shows text on the screen, or a Button component that you tap to initiate an action. Other components are more elaborate: a drawing Canvas that can hold still images or animations, an accelerometer (motion) sensor that works like a Wii controller and detects when you move or shake the phone, components that make or send text messages, components that play music and video, components that get information from Web sites, and so on.

The Designer window

Look at the Designer window. You'll see that it's divided into several areas:

- Toward the center is a white area called the Viewer. This is where you place components and arrange them. At the moment, it should be showing a label with
the same text as on the phone. This might not look exactly like it does on the phone, for example, the line of text might break at a different place. As you build your app, the Designer shows only rough indication of how it will look. To see how your app will really appear, look at the phone.

- To the left of the Viewer is the Palette, which is a list of components that you can select from. The Palette is divided into sections; currently only the Basic components are visible. You can see components in other sections of the Palette by clicking on the headers (in green).
- To the right of the Viewer is the Components list, which shows the components in your project. Compare this to the Palette on the left, which shows all the components available in App Inventor. The project at this point has only one component listed: Screen1, which represents the phone screen itself.
- Under the Components list is an area that shows the Media (pictures and sound) in the project. This project doesn't have any media yet, but you'll be adding some soon.
- At the far right is a section that shows the Properties of components. Properties are details about the components: color, text, font, and so on. Right now it's showing the properties of the screen, whose only properties are a background color, a background image, and a title. You'll use the properties in a moment to change the label.

You'll need a picture of the kitty and also the meow sound. Download these to your computer now and put them on your desktop, or in some other convenient location:

- kitty picture
- meow sound

Making the label

The first component to add is a Label:

1. Go to the Palette and click on Label (at about 5 down in the list of components) and drag it on to the Viewer. You'll see a rectangular shape appear on the Viewer, with the words Text for Label1. After a few seconds, the same text should appear on the phone screen.

2. Look at the Properties box at the right of the screen. It shows the properties of the label. There's a property called Text about halfway down, with a box for the label's text. Change the text to read "Pet the Kitty" and press return. You'll see the text change in the Designer and also on the phone.

3. Change the BackgroundColor of the label by clicking on the box, which currently reads "None", to select a color from the list that appears. Also change the TextColor of the label. Finally change the FontSize to 20. All these changes appear both in the Designer and on the phone, although the Designer view might not look exactly like the phone.

In App Inventor, you build the application on the phone right along with picking the components in the Designer. That way, you can see right away how your application will look. This Instant App Construction also applies to the behaviors you create for the components, as you'll learn below.

Adding the button

Go to the Palette and click on Button (at the top of the list of components) and drag it on to the Viewer, placing it below the label. You'll see a rectangular button shape appear on the Viewer. After about 10 seconds, a button should appear on the phone. You can go ahead and tap the phone button, but nothing will happen.
That's because your app hasn't told the button to do anything yet. Now make the button look like the kitty:

1. The Properties should be showing the properties of the button. If it isn't, click on the image of the button in the Viewer to expose the button's properties. In the Properties box, click on the area after Image (currently "None..."). A box appears with a button marked Add...

2. Click on Add... and you'll see Upload file.... Click on Choose File and browse to select the kitty.png file you downloaded to your computer earlier, and click OK.

3. You'll see a yellow message at the top of the screen: "Uploading kitty.png to the AppInventor server". After about 30 seconds the message and the upload box disappear, and kitty.png is listed as the image property for the button. You'll also see this listed in the Media area on of the design window. And if you look at the phone, you'll see the kitty picture has appeared in place of the button. The picture might not also appear in the Viewer. Don't worry if it doesn't.

4. If you look closely at the phone screen, you'll see that the kitty picture has the button text "Text for button 1" showing over it. You probably don't want that in your app, so change the text property of Button1 to something more appropriate, like "Pet me". Or just make it blank.

Tip: If the kitty.png file fails to upload from your computer to App Inventor, you'll see a red error. Try the upload again. If the file uploads, but the kitty picture doesn't appear on the phone, please try disconnecting and reconnecting the phone. If that doesn't work, look at the hints in the section on Troubleshooting. If the kitty appears on the phone, but the image on the designer remains as just the text, don't worry about it; that behavior is not yet fully implemented.

Adding the meow sound

Now make the kitty meow when you tap the button (that is, the picture of the kitty). First you need to have the meow sound:

1. Go to the Palette at the left of the browser window and click on the header marked Media to expand the Media section of the palette of components. Drag out a Sound component and place it in the Viewer. Wherever you drop it, it will appear in the area at the bottom of the Viewer marked Non-visible components.

2. Click on Sound1 to show its properties. Set its Source to meow.mp3. You'll need to follow the same steps to upload this file from your computer as you did to get the kitty picture. When you're done, you should see both kitty.png and meow.mp3 listed in the Media section of the Designer.
Making the sound play

Now make the kitty meow when you tap the button:

1. Look at the Blocks Editor window. (It was probably covered up by the Designer, so click it to make it fully visible.) The Blocks Editor is where you tell the components what to do, and when to do it. You're going to tell the kitty button to play a sound when the user taps it. If components are ingredients in a recipe, you can think of blocks as the cooking instructions.

   At the top left of the window, you'll see buttons labeled Built-in and My Blocks. Click on My Blocks, and you'll see a column that includes a drawer for each component: Button1, Label1, Screen1, and Sound1.

2. Click the drawer for the button. The drawer opens showing a selection of pieces that you can use to tell the button what to do, starting with when Button1.Click at the top.
3. Click on the block labeled `when Button1.Click` and drag it out into the workspace. (When you're looking for the block, notice that the word `when` is smaller than `Button1.Click`.)

4. Click on `Sound1` in My Blocks to open the drawer for the sound component, and drag out the `call Sound1.Play` block, fitting it into the gap marked `do` in the `when Button1.Click` block. The two blocks should snap together to form a unit, and you should hear a snapping sound.
5. Tap the button on the phone. You should hear the kitty meow. Congratulations! Your first app is running!

Tip: If the sound does not play, please use the No meow for Hello Purr Bug Report Form. This will lead you through some troubleshooting steps and also send us information that we can use to help isolate the causes of this problem.

Saving your app

To finish, try one more thing: Unplug your phone from the computer and then plug the phone back in. On your computer, go to the Blocks Editor and click Connect to Phone. After about 30 seconds, the phone app restarts with your work restored. App Inventor remembered your project and shows it on the phone. Even if you log out and come back later, your project will be saved.

When an app is initially built, it works only when the phone is connected to the computer, using some magic from App Inventor. In part 2 of this tutorial, you’ll teach the kitty new tricks and also learn how to get the app working as a self-contained application that works even when the phone is not connected.

Review

Here are the key ideas covered in this tutorial:

- You build apps by selecting components (ingredients) and then telling them what to do and when to do it.
- You use the Designer to select components. Some components are visible and some aren’t.
- You use the Blocks Editor to assemble blocks that define the components’ behavior
  - [blocks define event handlers, that tell components what to do when something happens.]
  - [blocks tell components to do things.]
- You can add media (sounds and images) to apps by uploading them from your computer.
Learn about App Inventor

Getting Started

Connect to the App Inventor web site and start a new project. Name it MoleMash, and also set the screen's Title to MoleMash. Open the Blocks Editor and connect to the phone.

Also download this picture of a mole and save it on your computer.

Introduction

You'll design the game so that the mole moves once every half-second. If it is touched, the score increases by one, and the phone vibrates. Pressing restart resets the score to zero.

This tutorial introduces:

- image sprites
- timers and the Clock component
- procedures
- picking random numbers between 0 and 1
- text blocks
- typeblocking

The first components

Several components should be familiar from previous tutorials:

- A Canvas named MyCanvas. This is the area where the mole moves.
- A Label named ScoreLabel that shows the score, i.e., the number of times the player has hit the mole.
- A Button named ResetButton

Drag these components from the Palette onto the Viewer and assign their names. Put MyCanvas on top and set its dimensions to 300 pixels wide by 300 pixels high. Set the Text of ScoreLabel to Score: --. Set the Text of ResetButton to Reset. Also add a Sound component and name it Noise. You'll use Noise to make the phone vibrate when the mole is hit, similar to the way you made the kitty purr in HelloPurr.

Timers and the Clock component

You need to arrange for the mole to jump periodically, and you'll do this with the aid of a Clock component. The Clock component provides various operations dealing with time, like telling you what the date is. Here, you'll use the component as a timer that fires at regular internals. The firing interval is determined by the Clock's TimerInterval property. Drag out a Clock component; it will go into the non-visible components area. Name it MoleTimer. Set its TimeInterval to 500 milliseconds to make the mole move every half second. Make sure that Enabled is checked.

Adding an Image Sprite

To add the moving mole we'll use a sprite.

Sprites are images that can move on the screen within a Canvas. Each sprite has a speed and a heading, and also an interval that determines how often the sprite moves at its designated speed. Sprites can also detect when they are touched. In MoleMash, the mole has a speed zero, so it won't move by itself. Instead, you'll be setting the mole's position each time the timer fires. Drag an ImageSprite component onto the Viewer. You'll find this component in the Animation category of the Palette. Place it within MyCanvas area. Set these properties for the Mole sprite:

- Picture: Use mole.png, which you downloaded to your computer at the beginning of this tutorial.
- Enabled: checked
- Interval: 500
- Heading: 0
- Speed: 0.0
- Visible: checked
- Width: Automatic
- Height: Automatic

You should see the X and Y properties already filled in. They were determined by where you placed the mole when you dragged it onto MyCanvas. Go ahead and drag the mole some more. You should see X and Y change. You should also see the mole on your connected phone, and the mole moving around on the phone you drop it around in the Designer. You've now specified all the components. The Designer should look like this. Notice how Mole is indented under MyCanvas in the...
Component Behavior and Event Handlers

Now you'll specify the component behavior. This introduces some new App Inventor ideas. The first is the idea of a procedure.

A procedure is a sequence of statements that you can refer to all at once as single command. If you have a sequence that you need to use more than once in a program, you can define that as a procedure, and then you don't have to repeat the sequence each time you use it. Procedures in App Inventor can take arguments and return values. This tutorial covers only the simplest case: procedures that take no arguments and return no values.

Define Procedures

Define two procedures:

- **MoveMole** moves the Mole sprite to a new random position on the canvas.
- **UpdateScore** shows the score, by changing the text of the ScoreLabel

Start with **MoveMole**:

1. In the Blocks Editor, under Built-In, open the Definition drawer. Drag out a procedure block and change the label procedure to **MoveMole**.

   **Note:** There are two similar blocks: procedure and procedureWithResult. Here you should use procedure.

   The **procedure** block has a slot labeled do. That's where you put the statements for the procedure. In this case there will be two statements: one to set the mole’s x position and one to set its y position. In each case, you’ll set the position to be a random fraction, between 0 and 1, of the difference between the size of the canvas and the size of the mole. You create that value using blocks for random-fraction and multiplication and addition. You can find these in the Math drawer.

2. Build the **MoveMole** procedure. The completed definition should look like this:

   **Leave the arg socket for MoveMole empty because MoveMole does not take any arguments. Observe how the blocks connect together:** The first statement uses the **Mole.X** block to set mole’s horizontal position. The value plugged into the block’s socket is the result of multiplying:
   
   1. the result of calling **random-fraction** block, which a value between 0 and 1
   2. the result of subtracting the mole's width from the canvas's width.

   The vertical position is handled similarly.
With MoveMole done, the next step is to define a variable called score to hold the score (number of hits) and give it initial value 0. Also define a procedure UpdateScore that shows the score in ScoreLabel. The actual contents to be shown in ScoreLabel will be the text "Score: " joined to the value of the score.

- To create the "Score: " part of the label, drag out a [text] block from the Text drawer. Change the block to read "Score: " rather than "text".
- Use a [join] block to attach this to a block that gives the value of the score variable. You can find the [join] block in the Text drawer.

Here's how score and UpdateScore should look:

Add a Timer

The next step is to make the mole keep moving. Here's where you'll use MoleTimer. Clock components have an event handler called [when ... Timer] that triggers repeatedly at a rate determined by the TimerInterval.

Set up MoleTimer to call MoveMole each time the timer fires, by building the event handler like this:

Notice how the mole starts jumping around on the phone as soon as you define the event handler. This is an example of how things in App Inventor start happening instantaneously, as soon as you define them.

Add a Mole Touch Handler

The program should increment the score each time the mole is touched. Sprites, like canvases, respond to touch events. So create a touch event handler for Mole that:

1. Increments the score.
2. Calls UpdateScore to show the new score.
3. Makes the phone vibrate for 1/10 second (100 milliseconds).
4. Calls MoveMole so that the mole moves right away, rather than waiting for the timer.

Here's what this looks like in blocks. Go ahead and assemble the Mole.Touched blocks as shown.

Here's a tip: You can use typeblocking: typing to quickly create blocks.

- To create a value block containing 100, just type 100 and press return.
- To create a MoveMole block, just type MoveMole and select the block you want from the list

Reset the Score

One final detail is resetting the score. That's simply a matter of making the Reset button change the score to 0 and calling UpdateScore.

Complete Program
Here's the complete MoleMash program:

Variations

Once you get the game working, you might want to explore some variations. For example:

- Make the game vary the speed of the mole in response to how well the player is doing. To vary how quickly the mole moves, you'll need to change the MoleTimer’s Interval property.
- Keep track of when the player hits the mole and when the player misses the mole, and show a score with both hits and misses. To do this, you'll need do define touched handlers both for Mole, same as now, and for MyCanvas. One subtle issue, if the player touches the mole, does that also count as a touch for the Canvas? The answer is yes. Both touch events will register.

Review

Here are some of the ideas covered in this project:

- Sprites are touch-sensitive shapes that you can program to move around on a Canvas.
- The Clock component can be used as a time to make events that happen at regular intervals.
- Procedures are defined using to blocks.
- For each procedure you define, App Inventor automatically creates an associated call block and places it in the My Definitions drawer.
- calling random-fraction produces a number between 0 and 1
- text blocks specify literal text, similar to the way that number blocks specify literal numbers.
- typeblocking is a way to create blocks quickly, by typing a block’s name.
App Inventor Capabilities and Limitations

David Wolber

You can build many apps with App Inventor, but there are limitations. Most of the limitations are not inherent to App Inventor’s design, but instead related to the immaturity of the tool-- the App Inventor team hasn’t yet implemented the functionality, but will.

Capabilities

App Inventor’s capabilities include:

- Access to most of the phone’s functionality: phone calls, SMS texting, sensors for location, orientation, and acceleration, text-to-speech and speech recognition, sound, video.
- The ability to invoke other apps, with the ActivityStarter component
- Programming control just as with a textual language. There are blocks for conditionals (if, ifelse), foreach, and while, and a fairly comprehensive list of math and logic blocks.
- Database access, both on the device and on the web. So you can save data persistently, and with a web database share data amongst phones.
- Access to web information sources (APIs)-- you can bring in data from Facebook, Amazon, etc. See limitations below.

Limitations

App Inventor has the following limitations in terms of the apps you can build:

- **Limited UIs.** The user interface builder has improved but is still a bit buggy and limited, so you can’t build any user interface. For instance, you can’t create apps with multiple screens and handling orientation changes has some glitches. These problems are not fundamental to the design of App Inventor and will soon be fixed.
- **Limited Access to the device.** There are not yet components for all the data and functionality of the phone. For instance, you can’t save and retrieve files from the file system and you have only limited access to the contact list (e.g., you cannot create groups).
- **Limited Access to Web.** You can only access APIs that follow a particular protocol (App-Inventor-compatible APIs). So if you want to get data from the web, you’ll need to program or have a programmer create an App-Inventor-Compliant API that wraps an existing API.
- **No polymorphic components.** Function blocks are tied to specific components, so there is no way to call functions on a generic component. For instance, if you create a procedure MoveXY, it has to be tied to a specific image sprite, not a general image sprite.
- **Limited access to the Android Market.** The apps (.apk files) generated by App Inventor lack the required configuration for direct inclusion in the market. However, there is now a workaround for market publication. See instructions at http://www.androidworld.it/forum/app-inventor-91/%5Bhow-%5D-publish-app-created-appp-inventor-android-market-4597/.
Strategies for Success in Teaching App Inventor

David Wolber

Following are the strategies I've used in my course "Computing, Robots, and the Web", which qualifies as a Math core course at USF. The students are primarily humanities, science, and business students.

**Portfolios and student-owned accounts**

On the first day of class, students register for Google Sites accounts and create a portfolio site. From that day on they post everything they do-- from small lab assignments to large, creative projects-- on their portfolio. I tell them that if its not on their portfolio, it doesn't exist! I also assign students their own App Inventor accounts the first day of class, as opposed to using generic class accounts. These are the advantages of the portfolio approach:

- The motivation level is higher when students know they are creating something that their classmates and others can view.
- Students can continue to work on their projects when the course ends.
- Students can refer back to the samples and projects they've done previously.
- One outcome of the class is a portfolio of their work which they can show their family, friends, and prospective employers.
- Its a nice way to introduce and encourage "cloud computing".

Here's an example of a student portfolio via USF student MackenzieLisenbey: https://sites.google.com/site/lisenbyportfolio/drawing-lab.

**Worked-out samples and assigned variations**

Even more than with the typical CS major, CS 0 students learn better with concrete examples as opposed to abstract concepts. The students build apps following fully worked-out tutorials that explain the behaviors (blocks) each step of the way. They're then given assignments that ask them to program variations and additions on those samples. The sample apps become part of the vocabulary for the class. I find that "it's like the quiz example, when you step through the questions," works better than, "you know how iteration works, just increment the index variable..."

**Interesting apps and student-initiated learning**

The assigned sample apps should be interesting in terms of their end-result. Samples that just illustrate an interesting computer science concept don't work so well with this audience. Whereas CS students are motivated puzzle-solvers, the less technical student is motivated from creating something cool or useful to the world.

You can still get to the computer science concept, just not in a top-down, concept-first manner. As the students work on interesting samples, they invariably think of ideas for customization and other apps. "Hey, Wolber, how would I do this? What if I wanted to bring in my tweets? This quiz is cool, but how would I make a multiple choice quiz?" When they are motivated to solve a real-world problem you can teach them the concepts.

https://sites.google.com/site/appinventor/app-inventor-and-computer-science-for-non-cs
Creative Projects
Some students take off immediately as soon as they begin building the sample projects. The motivation level rises dramatically, however, when I assign the first creative project, and let the students create whatever they want. For some students, this is when they really buy in.

I've assigned two major projects each semester. Groups of two have worked best. People say that its best to group students of similar abilities-- I agree. I also require the teams to assign each member individual programming deliverables.

At the beginning of each creative project, the students are given time in class to explore their ideas and build a project page. They develop an "elevator pitch", perform some market research, and in general build a mini-business plan and specification for the app they're going to build. I require them to create a prototype early on and perform "user testing" with their friends and other students. This is all informal and fun, but it gives them an idea about how to take an idea from concept to reality. At some point, I'd like to develop some better lessons in this area and include readings such as Kawasaki's The Art of the Start.

Market/Studio for Publishing Apps
I created a USF Android Market for my Spring '10 course and the students were required to submit their mid-term and final projects there. The market provides another level of motivation for students-- they know that many people will visit it. The students post their projects, including a barcode that people can use to install their app, source code that can be uploaded into App Inventor for customization, and a distilled version of their "business plan"

The market is a Google Sites page with all students as "collaborators" I provided a template for submission so that there would be some uniformity in the site, though this didn't prove too successful. I think the potential for this is great: I envision is a well-designed app studio that can be promoted throughout the university.

App Inventor and CS 0 Core Requirements
The requirements for a core-curriculum computer science course vary from school to school. I've integrated "Internet and Society" readings (e.g., The Big Switch), web design, and web 2.0 tools as part of my course. App Inventor itself is a great vehicle for teaching programming concepts, web services, GPS, web 2.0, and just about any other computing concept.

There are also ample opportunities for the students to learn math concepts-- I could even envision App Inventor being a tool for a Math course. Location-aware apps are great for this,e.g., a tour guide app that gives the user different information based on what building/monument is nearby. For such apps, students need to calculate distance based on two GPS coordinates, calculate whether or not the phone is within a given boundary, where the boundary is a rectangle or even an arbitrary polygon. The motivation level for solving such problems, in the context of building an app, is out the roof.
Quiz Me

What You're Building

QuizMe is a trivia game about baseball, but you can use it as a template to build quizzes on any topic. With QuizMe:

- The user steps through a series of questions, clicking a button to proceed to the next question.
- The user enters an answer for each question and the app reports whether each answer is correct or not.

With QuizMe, the quiz questions are always the same unless you, the programmer, change them. Later, you can create MakeQuiz, an app that lets users of the app create and modify the quiz questions.

This tutorial assumes you are familiar with the basics of App Inventor -- using the Component Designer to build a user interface, and using the Blocks Editor to specify event-handlers. If you are not familiar with the basics, try stepping through some of the basic tutorials before continuing.

Getting Started

Connect to the App Inventor web site and start a new project. Name it QuizMe, and also set the screen's Title to QuizMe. Open the Blocks Editor and connect to the phone.

Also download these pictures of baseball players and save them on your computer: Larsenberra.jpg, dallasbraden.jpg, cyyoung.jpg, nolanryan.jpg. Later, you'll load these images into your project.

Introduction

You'll design the quiz game so that the user proceeds from question to question by clicking a Next button, and receives simple correct/incorrect feedback on each answer.

This tutorial introduces:

- Defining and displaying lists of information
- Sequencing through a list using an index variable -- a variable that keeps track of a position in a list.
- Conditional behaviors-- performing certain operations only when a condition is met.
- Switching an image to show a different picture at different times.

Set up the Components

Use the component designer to create the interface for QuizMe. When you finish, it should look something like the snapshot below (there are also more detailed instructions below the snapshot).
To create this interface, first load the images you downloaded into the project. Click on the Add button in the Media area and select one of the downloaded files (e.g., Larsenberra.jpg). Then do the same for the other three images.

Next, create the following components by dragging them from the Palette into the Viewer.

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Palette Group</th>
<th>What you'll name it</th>
<th>Purpose of Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Basic</td>
<td>Image1</td>
<td>The picture part of the question</td>
</tr>
<tr>
<td>Label</td>
<td>Basic</td>
<td>QuestionLabel</td>
<td>Displays the current question</td>
</tr>
<tr>
<td>HorizontalArrangement</td>
<td>Screen Arrangement</td>
<td>HorizontalArrangement1</td>
<td>Organizes the AnswerPrompt and Text</td>
</tr>
<tr>
<td>Label</td>
<td>Basic</td>
<td>AnswerPromptLabel</td>
<td>Text prompting for an answer</td>
</tr>
<tr>
<td>TextBox</td>
<td>Basic</td>
<td>AnswerText</td>
<td>User will enter answer here.</td>
</tr>
<tr>
<td>Label</td>
<td>Basic</td>
<td>RightWrongLabel</td>
<td>Correct/Incorrect is displayed here.</td>
</tr>
<tr>
<td>HorizontalArrangement</td>
<td>Screen Arrangement</td>
<td>HorizontalArrangement2</td>
<td>Organizes the AnswerButton and NextButton</td>
</tr>
<tr>
<td>Button</td>
<td>Basic</td>
<td>AnswerButton</td>
<td>User clicks to submit an answer</td>
</tr>
<tr>
<td>Button</td>
<td>Basic</td>
<td>NextButton</td>
<td>User clicks to proceed to the next answer</td>
</tr>
</tbody>
</table>

Set the properties of the components as described below:

<table>
<thead>
<tr>
<th>Component</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image1</td>
<td>Set its Picture property to Larsenberra.jpg. This is the first picture that should appear.</td>
</tr>
<tr>
<td>QuestionLabel</td>
<td>Change Text property to &quot;question&quot;</td>
</tr>
<tr>
<td>AnswerPromptLabel</td>
<td>Change Text property to &quot;Enter Answer&quot;. On Viewer screen, move this label into HorizontalArrangement1.</td>
</tr>
<tr>
<td>AnswerText</td>
<td>Change Hint to &quot;please enter an answer&quot;. On Viewer, move AnswerText into HorizontalArrangement1.</td>
</tr>
<tr>
<td>AnswerButton</td>
<td>Change Text property to &quot;Submit&quot;. On Viewer, move the button into HorizontalArrangement2.</td>
</tr>
<tr>
<td>NextButton</td>
<td>Change Text property to &quot;Next&quot;. Move the button into HorizontalArrangement2.</td>
</tr>
<tr>
<td>RightWrongLabel</td>
<td>Change Text property to &quot;correct/incorrect&quot;</td>
</tr>
</tbody>
</table>

Add behaviors to the components

Open the Blocks Editor to add the behaviors for the app. First, you'll define two list variables, QuestionList to hold the list of questions, and AnswerList to hold the list of corresponding answers.

To define the two list variables, you'll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>def variable</td>
<td>Definitions</td>
<td>Defines the QuestionList variable (rename it)</td>
</tr>
<tr>
<td>def variable</td>
<td>Definitions</td>
<td>Defines the AnswerList variable (rename it)</td>
</tr>
<tr>
<td>make a list</td>
<td>Lists</td>
<td>Used to insert the items of the QuestionList</td>
</tr>
</tbody>
</table>
You create global variables by dragging in a def variable block from the Definitions drawer and double-clicking the default name "variable" to change its name. The def variable block has a slot for the initial value of the variable. If the variable represents a number or text, click the triangle on the red ? block and choose the type you want. If the variable represents a list, drag the red ? block into the trash, go to the Lists drawer, and plug in a make a list block into the variable definition.

The blocks should look like this:

**Define the Hidden Index Variable**

Each time the user clicks the NextButton to proceed through the quiz, the app needs to remember which question it is on. In programming, to remember something, you define a new variable. In this case, the app needs to remember the current question number — the index into the list QuestionList.

To create the variable currentQuestionIndex, you'll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>def variable</td>
<td>Definitions</td>
<td>Defines the currentQuestionIndex variable (rename it)</td>
</tr>
<tr>
<td>number (1)</td>
<td>Math</td>
<td>Set the initial value of currentQuestionIndex to 1</td>
</tr>
</tbody>
</table>

The blocks should look like this:

**Display the First Question**

To start, you'll ignore the answers and just work on the behavior to sequence through the questions. The desired behavior is the following: when the app starts, the first question should appear in the label named QuestionLabel. When the user clicks the NextButton, the second question should appear. When the user clicks again, the third should appear. When the last question is reached, clicking the NextButton should result in the first question once again appearing in the QuestionLabel.

With App Inventor, you select particular items in a list with the select list item block. The block asks you to specify the list and an index—a position in the list. If a list has three items, the indexes 1,2, and 3 are valid.

For QuizMe, when the app starts, the app should choose the first question in the list and display it in the QuestionLabel component.

For this app initialization behavior, you'll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen1.Initialize</td>
<td>Screen1</td>
<td>When app begins this event-handler is triggered.</td>
</tr>
<tr>
<td>set QuestionLabel.Text to</td>
<td>QuestionLabel</td>
<td>Need to put the first question in QuestionLabel</td>
</tr>
<tr>
<td>select list item</td>
<td>Lists</td>
<td>Need to select the first question from QuestionList</td>
</tr>
<tr>
<td>global QuestionList</td>
<td>My Definitions</td>
<td>The list to select from</td>
</tr>
<tr>
<td>number (1)</td>
<td>Math</td>
<td>select the first question by using an index of 1</td>
</tr>
</tbody>
</table>
The blocks should look like this:

How the Blocks Work

The `Screen1.Initialize` event is triggered when the app begins. The first item of the variable `QuestionList` is selected and placed into `QuestionLabel.Text`. So when the app begins, the user will see the first question.

Test this behavior. Click `Restart Phone App` (or `Connect Phone` if not connected). What appears on the phone? If you created the `QuestionList` as described above, the first item of `QuestionList`, "Who pitched a perfect game in the World Series?", should appear in the `QuestionLabel`.

Iterating Through the Questions

Now program the behavior of the `NextButton`. You’ve already defined the `currentQuestionIndex` to remember the question the user is on. When `NextButton` is clicked, the app needs to increment this variable, e.g., change it from 1 to 2 or from 2 to 3, etc., and then use the resulting value to select the new "current" question. For this behavior, you’ll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextButton.Click</td>
<td>NextButton</td>
<td>When user clicks Next, this event-handler is triggered.</td>
</tr>
<tr>
<td>set currentQuestionIndex to</td>
<td>My Definitions</td>
<td>Need to put the first question in <code>QuestionLabel</code></td>
</tr>
<tr>
<td>+ block</td>
<td>Math</td>
<td>Used to increment <code>currentQuestionIndex</code></td>
</tr>
<tr>
<td>global currentQuestionIndex</td>
<td>My Definitions</td>
<td>New value will be old value + 1</td>
</tr>
<tr>
<td>number (1)</td>
<td>Math</td>
<td>For the + 1</td>
</tr>
<tr>
<td>set QuestionLabel.Text to</td>
<td>QuestionLabel</td>
<td>Need to display the next question here</td>
</tr>
<tr>
<td>select list item</td>
<td>Lists</td>
<td>Need to select the first question from <code>QuestionList</code></td>
</tr>
<tr>
<td>global QuestionList</td>
<td>My Definitions</td>
<td>Plug into list slot of call select list item</td>
</tr>
<tr>
<td>global currentQuestionIndex</td>
<td>My Definitions</td>
<td>Plug into index slot of call select list item, we want nth item</td>
</tr>
</tbody>
</table>

The blocks should look like this:

How the Blocks Work

The first row of blocks increments the variable `currentQuestionIndex`. If `currentQuestionIndex` has a 1 in it, it is changed to 2. If it has a 2, it is changed to 3, and so on. Once the `currentQuestionIndex` variable has been changed, the app uses it to select the "current" question.

Recall that in the `Screen.Initialize` event-handler, the app selected the first question to display:

When the `NextButton` is clicked, the app doesn't choose the first item in the list, or the 2nd or 3rd, it chooses the `currentQuestionIndex`th item.
The blocks are executed in a right-to-left manner. The app first evaluates the `index` parameter of `select list item`, which is the variable `currentQuestionIndex`. The number is stored in `currentQuestionIndex` is used as the index when the select list item is executed.

When the `NextButton` is clicked for the first time, the increment blocks will set `currentQuestionIndex` from 1 to 2, so the app will select the second item from `QuestionList`, "who pitched the first perfect game of 2010?". The second time `NextButton` is clicked, `currentQuestionIndex` will be set from 2 to 3, and the app will select the 3rd question in the list, "who pitched the first perfect game of the modern era?"

Test this behavior. Test the behavior of the `NextButton` to see if the app is working correctly thus far. To test, play the role of the user and click the `NextButton` on the phone. Does the phone display the second question, "Who pitched the first perfect game of 2010?" It should, and the third question should appear when you click the `NextButton` again. If this is working, pat yourself on the back quickly, and then go on.

Try clicking the `NextButton` again (a third time). You should see an error: "Attempting to get item 4 of a list of length 3". The app has a bug—do you know what the problem is?

The problem with the app is that it always increments the `currentQuestionIndex` variable when the `NextButton` is clicked. When `currentQuestionIndex` is already 3 and the user clicks the `NextButton`, the app changes `currentQuestionIndex` from 3 to 4, then calls `select list item` to get the `currentQuestionIndex`th, or in this case, the 4th item. Since there are only three items in the variable `QuestionList`, Android complains.

What the app needs to do is ask a question-- check a condition-- when the `NextButton` is clicked, and execute different blocks depending on the answer. One way to ask the question is to ask, "is the variable `currentQuestionIndex` already 3?" If the answer is yes, you should set `currentQuestionIndex` back to 0 so the user is taken back to the first question.

You'll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if test then-do</code></td>
<td>Control</td>
<td>To ask if user is on last question</td>
</tr>
<tr>
<td><code>=</code> block</td>
<td>Math</td>
<td>to test if <code>currentQuestionIndex</code> is 3</td>
</tr>
<tr>
<td><code>global currentQuestionIndex</code></td>
<td>My Definitions</td>
<td></td>
</tr>
<tr>
<td><code>number 3</code></td>
<td>Math</td>
<td>3 is number of items in the list</td>
</tr>
<tr>
<td><code>set currentQuestionIndex to</code></td>
<td>My Definitions</td>
<td>set to 0 to go back to first question</td>
</tr>
<tr>
<td><code>number (0)</code></td>
<td>Math</td>
<td>set to 0 because next blocks will increment to 1</td>
</tr>
</tbody>
</table>

The modified `NextButton.Click` event-handler should look like this:

How the Blocks Work

When the `NextButton` is clicked, the app first checks to see if `currentQuestionIndex` has a 3 in it. If it does, `currentQuestionIndex` is set back to 0 so that when 1 is added to it with the blocks below, it will be 1 and the quiz will loop back to display the first question. Note that only the blocks inset within the `if-test-then-do` block are dependent on the condition-- the increment and `set QuestionLabel.Text` to blocks are executed under all conditions.

Test this behavior. Because variables like `currentQuestionIndex` are hidden, they are often the source of bugs in a program. Fortunately, App Inventor provides a way to make such hidden variables transparent during testing. Specifically, App Inventor allows you to "watch" how the value of a variable change as an app progresses. For this test, right-click the `currentQuestionIndex` def block.
in the Blocks Editor and choose Watch. Then choose **Restart Phone App**. The def block will then appear with a watch box showing the initial value of `currentQuestionIndex` (1):

Now pick up the phone and click the NextButton. The second question, "who pitched the most recent perfect game in the major leagues?" should appear in the QuestionLabel on the phone, as before. On the App Inventor screen, 2 should appear in the `currentQuestionIndex` memory cell:

When you click again, the third question should appear on the phone and 3 should appear in the memory cell. If you click again, 1 should appear in `currentQuestionIndex` and the first question on the phone.

### A Maintainable App: Making it Easy to Modify the Questions

Next, you'll modify the app to make it easy to add and remove elements from the list. You'll rewrite the blocks so that they'll work on any list, not just one with exactly three items. To begin, add a fourth question to `QuestionList` and another answer into `AnswerList`. The blocks should look like this:

Test the modified app. Click the **NextButton** a number of times. You should see that the fourth question never appears, no matter how many times you click Next.

The problem is that the test to see if the user is on the last question is too specific: it asks if the `currentQuestionIndex` variable is 3:

You could just change the number 3 to a 4, and the app would again work correctly. The problem with this solution, however, is that each time you modify the questions and answers, you also have to remember to make this change. Such dependencies in a computer program often lead to bugs, especially as an app grows in complexity. It's much better to set the program up so that it will work no matter how many questions there are. Such generality is even more important when the list you are working with changes dynamically, e.g., a quiz app that allows the user to add new questions.

The better solution is to ask the question in a more general way. You really want to know if the current question the user is on-- the value of `currentQuestionIndex`--
is as large as the number of items in `QuestionList`. If the app asks the question in this more general manner, it will work even when you add to or remove items from the `QuestionList`. To modify the `NextButton.Click` event-handler you'll replace the previous test that referred directly to 3. You'll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>length of list</code></td>
<td><code>Lists</code></td>
<td>asks how many items are in <code>QuestionList</code></td>
</tr>
<tr>
<td><code>global QuestionList</code></td>
<td><code>My Definitions</code></td>
<td>put into list slot of length of list</td>
</tr>
</tbody>
</table>

Your `NextButton.Click` event-handler should now appear as:

![Diagram of modified NextButton.Click event-handler]

**How the Blocks Work**

The if test now compares the `currentQuestionIndex` to the length of the `QuestionList`. So if `currentQuestionIndex` has a 4 in it, and the length of the `QuestionList` is 4, then the `currentQuestionIndex` will be set to 0 (and then 1 after the increment operation in the first row of blocks after the if). Note that, because the blocks no longer refer to 3 or any specific size, the behavior will work no matter how many items are in the list.

Test the modified behavior. When you click the `NextButton`, does the app now sequence through the four questions, moving to the first one after the fourth?

**Switching the Image for Each Question**

The current app shows the same image, no matter what question is being asked. You can change this so an image pertaining to each question appears when the `NextButton` is clicked. Earlier, you added four pictures as media for the project. Now, you'll create a third list, `PictureList`, with the names of the image files as its items. and you'll modify the `NextButton.Click` event-handler to switch the picture each time.

First, create a `PictureList` and initialize it with the names of the image files. Be sure that the names are exactly the same as the names of the files that were loaded in to the media of the project. Here's how the blocks for the `PictureList` should look:

![Diagram of PictureList]

Next, you need to modify the `NextButton.Click` event-handler so that it modifies the picture depending on what question the user is on. If you set the `Image.Picture` property to a file name of an image that has been loaded, that image will appear. To modify `NextButton.Click`, you'll need the following blocks:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set Image1.Picture to</code></td>
<td><code>Image1</code></td>
<td>set this to change the picture</td>
</tr>
<tr>
<td><code>select list item</code></td>
<td><code>Lists</code></td>
<td>need to select the picture corresponding to current question</td>
</tr>
<tr>
<td><code>global PictureList</code></td>
<td><code>My Definitions</code></td>
<td>select a file name from this list</td>
</tr>
<tr>
<td><code>global currentQuestionIndex</code></td>
<td><code>My Definitions</code></td>
<td>select the <code>currentQuestionIndex</code>th item</td>
</tr>
</tbody>
</table>

Here is how the blocks should look:
How the Blocks Work

The \texttt{currentQuestionIndex} serves as the index for the \texttt{QuestionList} and the \texttt{PictureList}. When \texttt{currentQuestionIndex} is 1, the app selects the first question and the first picture. When \texttt{currentQuestionIndex} is 2, the app selects the second question and second picture. Of course this scheme depends on the lists being in sync and indeed they are. For instance, the first picture, \texttt{LarsenBerra.jpg} is a picture of Don Larsen, and Don Larsen is the answer to the first question, “Who pitched a perfect game in the World Series?” Test the modified behavior. Does a different image appear each time you click the NextButton?

Evaluating Answers

Next, you’ll add blocks that report whether the user has answered a question correctly or not. The user enters the answer in \texttt{AnswerText} and then clicks the \texttt{AnswerButton}. The app must compare the user’s entry with the answer to the “current” question, using an \texttt{ifelse} block to check. The \texttt{RightWrongLabel} should be modified to report whether or not the answer is correct. You’ll need the following blocks for this behavior:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{AnswerButton.Click}</td>
<td>\texttt{AnswerButton}</td>
<td>the behavior is triggered when user clicks the \texttt{AnswerButton}</td>
</tr>
<tr>
<td>\texttt{ifelse}</td>
<td>\texttt{Control}</td>
<td>if answer is correct, do one thing, else do another</td>
</tr>
<tr>
<td>\texttt{=} block</td>
<td>\texttt{Math}</td>
<td>to ask if answer is correct</td>
</tr>
<tr>
<td>\texttt{AnswerText.Text}</td>
<td>\texttt{AnswerText}</td>
<td>the user’s answer is in this textbox</td>
</tr>
<tr>
<td>\texttt{select list item}</td>
<td>\texttt{Lists}</td>
<td>to select the current answer from \texttt{AnswerList}</td>
</tr>
<tr>
<td>\texttt{global AnswerList}</td>
<td>\texttt{My Definitions}</td>
<td>The list to select from</td>
</tr>
<tr>
<td>\texttt{global currentQuestionIndex}</td>
<td>\texttt{My Definitions}</td>
<td>the question number (and answer number) the user is on</td>
</tr>
<tr>
<td>\texttt{set RightWrongLabel.Text to}</td>
<td>\texttt{RightWrongLabel}</td>
<td>report the answer here</td>
</tr>
<tr>
<td>\texttt{text “correct”}</td>
<td>\texttt{Text}</td>
<td>if answer is right</td>
</tr>
<tr>
<td>\texttt{set RightWrongLabel.Text to}</td>
<td>\texttt{RightWrongLabel}</td>
<td>report the answer here</td>
</tr>
<tr>
<td>\texttt{text “incorrect”}</td>
<td>\texttt{Text}</td>
<td>if answer is wrong</td>
</tr>
</tbody>
</table>

The blocks should look like this:

How the Blocks Work

The \texttt{ifelse} test reads, “is the user’s answer (\texttt{AnswerText.Text}) equal to the \texttt{currentQuestionIndex}th item in the \texttt{AnswerList}?” If \texttt{currentQuestionIndex} is 1, the app will compare the user’s answer with the first item in \texttt{AnswerList}, “Don Larsen”. If \texttt{currentQuestionIndex} is 2, the app will compare the user’s answer with the second
answer in the list, "Dallas Braden", and so on. If the test result is positive, the then-do blocks are executed and the RightWrongLabel is set to "correct!". If the test is false, the else-do blocks are executed and the RightWrongLabel is set to "incorrect".

Test the modified app. Try answering one of the questions. It should report whether or not you answered the question exactly as is specified in the AnswerList. Test with both a correct and incorrect answer (because text is being compared, the test is case-sensitive).

Click the NextButton and answer a second question. Does it still work? It should, but you might notice that when you click the NextButton, the "correct!"/"incorrect" text and the previous answer are still there. Though it's fairly innocuous, such user interface issues will definitely be noticed by the users of your app.

To blank out the RightWrongLabel and the AnswerText, you'll put the following blocks within the NextButton.click event-handler:

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Drawer</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>RightWrongLabel.Text to RightWrongLabel</td>
<td>the label to blank out</td>
</tr>
<tr>
<td>text (blank)</td>
<td>Text</td>
<td>When Next is clicked, erase old answer critique</td>
</tr>
<tr>
<td>set</td>
<td>AnswerText.Text to AnswerText</td>
<td>the user's answer from previous question</td>
</tr>
<tr>
<td>text (blank)</td>
<td>Text</td>
<td>When Next is clicked, erase old answer</td>
</tr>
</tbody>
</table>

The blocks should look like this:

![Block Diagram]

How the Blocks Work

When the NextButton is clicked, the user is moving on to the next question, so the top two rows of the event-handler blank out the RightWrongLabel and the AnswerText.

Test this behavior. Answer a question and click Submit, then click the NextButton. Does your previous answer and the app's critique disappear?

Final Program

QuizMe! Final Version
Package the final version of the app by choosing Package For Phone | Barcode from the Component Designer menu. When the barcode appears, use the barcode scanner on your phone to download and install the app.

Variations

Once you get a quiz working, you might want to explore some variations. For example,

- Instead of just showing images for each question, try playing a sound clip or a short video. With sound, you can turn your quiz app into a Name That Tune app.
- The quiz is very rigid in terms of what is accepted as a valid answer. There are a number of ways to modify this. One is to use the text.contains block to see if the user's answer is contained in the actual answer. Another is to provide multiple answers for each question, and check by iterating (foreach) through them to see if any match.
- Transform the quiz so that it is multiple-choice. The list of answers will need to be a list of lists, with each sub-list holding the answer choices. Use the ListPicker component to allow the user to choose an answer.

Review

Here are some of the ideas covered in this tutorial:

- Apps can be written in a general manner so that they work with any data list.
- Index variables are used to track the current position within a list. When you increment them, be careful about reaching the end of the list.

Google is grateful to Professor David Wolber, CS Professor at The University of San Francisco, for developing this tutorial.
Android, Where's My Car

You parked somewhere near the stadium or bar, but when the concert/party ends you don’t have a clue where the car is. The friends you came with are equally as clueless.

Fortunately you haven’t lost your Android phone that never forgets anything, and you remember you have the hot new app, Android, Where’s My Car?. With this app, you click a button when you park your car, and the Android uses its location sensor to record the car’s GPS coordinates and address. Later, when you reopen the app, it shows you a map from where you are to the remembered location—problem solved!

Download it, Customize it, Understand it

Below is a customizable, annotated and explained Android, Where’s My Car? app. You can:

- install the app on your phone by scanning a barcode
- download the source code blocks for the app, upload them into App Inventor, then customize the app or use it as the basis for another idea you have.
- Study the annotated blocks and explanations below to better understand the app and App Inventor programming in general.

TRY IT. Scan the barcode to your Android phone. It will step you through installing the app.

(If you don’t have a scanner, download the free ZXing scanner by searching for it at the Android Market)

CUSTOMIZE IT. Download the source code blocks to create your own customized app:

1. save the source file (.zip) to your computer (click icon to the right)-->
2. do not unzip it-- leave it as a zip file.
3. open the My Projects page in App Inventor.
4. select More Actions | Upload Source from the menu.
5. choose the file you just downloaded

Understand It

The app demonstrates how to communicate with the Android location sensor, how to record data in the phone’s long-term memory (database), and how you can open the Google Maps app from your app to show directions from one one location to another. It makes use of the following App Inventor components:

- Location Sensor
- TinyDB -- to store the data
- ActivityStarter -- to open a map

The User Interface

Here are the components for the Android, Where’s My Car? app, as shown in the Component Designer:
The user interface consists of labels to show location data and buttons to initiate events. Some labels just show static text, e.g., GPSLabel is the text “GPS:” that appears in the user interface. Others, such as CurrentLatLabel, will display dynamic data one the location sensor gets its readings. For these labels, a default value is set (0.0) here in the Component Designer.

The ActivityStarter1 component is used to launch the map when the user asks for directions. Its properties are only partially shown above. Here is how they should be specified:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>android.intent.action.VIEW</td>
</tr>
<tr>
<td>ActivityPackage</td>
<td>com.google.android.apps.maps</td>
</tr>
<tr>
<td>ActivityClass</td>
<td>com.google.android.maps.MapActivity</td>
</tr>
</tbody>
</table>

The App’s Behavior

Here are the blocks for the Android, Where's My Car? app (the yellow annotations will also appear when you load this app into App Inventor):
Let's examine the four different event-handlers of the app, starting in the top-left and working around in counter-clockwise order.

**LocationSensor1.LocationChanged.** This event occurs when the phone's location sensor first gets a reading, or when the phone is moved to produce a new reading. The event-handler just places the readings—latitude, longitude, and current (street) address—into the corresponding "Current" labels so that they appear on the phone. The **RememberButton** is also enabled in this event-handler. Its enabled setting should be unchecked in the Component Designer because there is nothing for the user to remember until the sensor gets a reading.

**RememberButton.Click.** When the user clicks the **RememberButton**, the location sensor's current readings are put into the "remember" labels and stored to the database as well. The **DirectionsButton** is enabled as it now makes sense for the user click on it to see a map (though it will make more sense once the user changes location).

**DirectionsButton.Click.** When the user clicks the **DirectionsButton**, the event-handler builds a URL for a map and calls **ActivityStarter** to launch the Maps application and load the map. **make text** is used to build the URL to send to the Maps application. The resulting URL consists of the Maps domain along with two crucial parameters, **saddr** and **daddr**, which specify the start and destination for the directions. For this app, the **saddr** is set to the latitude and longitude of the current location, and the **daddr** is set to the latitude and longitude of the location that was "remembered" (the location of your car!).

**Screen1.Initialize.** This event is always triggered when an app opens. To understand it, you have to envision the user recording the location of the car, then closing the app, then later re-opening the app. When the app re-opens, the user expects that the location remembered earlier should appear on the phone. To facilitate this, the event-handler queries the database (**TinyDB.GetValue**). If there is indeed a remembered address stored in the database— the length of the stored address is greater than zero—the remembered latitude, longitude, and street address are placed in the corresponding labels.

**Variations**
- Create "Android, Where is Everyone?", an app that lets a group of people track each other's whereabouts. Whether your hiking or at the park, this app could help save time and even lives.
- Create a "breadcrumb" app that tracks your (phone's) whereabouts by recording each location change. One interesting refinement would be to only record a new "breadcrumb" if the location has changed by a certain amount.

**Review**
Here are some of the ideas covered in this tutorial:
- The **LocationSensor** component can report the phone's latitude, longitude, and current street address. Its **LocationChanged** event is triggered when sensor gets its first reading and when the reading changes (the phone has moved).
- The **ActivityStarter** component can launch any app including Google Maps. For Maps, you set the **DataUrl** property to the URL of the map you want to display. If you want to show directions, the URL will be of the form: http://maps.google.com/maps/?saddr=0.1,0.1&daddr=0.2,0.2, where the numbers are GPS coordinates.
- *make text* is used to piece together (concatenate) separate text items into a single text object. It allows you to concatenate dynamic data with static text. With the Maps URL, the GPS coordinates are the dynamic data.
- *TinyDB* allows you to store data persistently in the phone's database. Whereas the data in a variable or property is lost when an app closes, the data you store in the database can be loaded into your app each time it is opened.

Google is grateful to *Professor David Wolber*, CS Professor at The University of San Francisco, for developing this tutorial.