MODULE ONE

INTRODUCTION TO PROGRAMMING AND APP INVENTOR

Module Objectives

At the end of this module, the student should be able to:

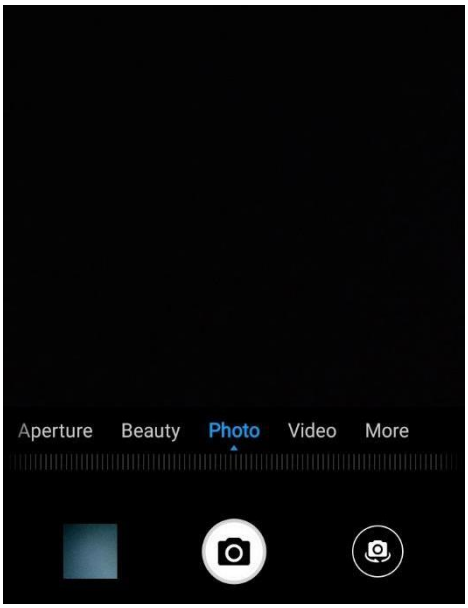
- **navigate easily around the MIT App Inventor platform.**
- **create an App Inventor project.**
- **design a basic Android app interface with App Inventor.**
- **run an Android app from App Inventor on an Android device and the emulator.**

The Connection between Android Devices and Applications

Everything generally done on an Android device, and on computing devices (which include computers, laptops, smartphones, tablets, smartwatches, etc.) are done through things called **applications**, or **apps** for short. An application is a piece of software that has been *programmed* to perform a specific function or group of functions, and these functions are executed when the app is *run*. On most computing devices, an application has a main icon which is what you see in your app menu, and clicking on this icon *runs* the application.

Let us take the camera app, for example. The main function of the camera app is to **take pictures**. At the most basic level, that is what every single camera application on any device does. Whenever you want to take pictures of your cat or dog, or a selfie with your friends, you look for the camera application app and *run* it by clicking on the icon. This is generally how you run an application on any device.

Now, what happens when you click on the camera icon to run the app? Another screen opens, which in most cases would look similar to the one below:



This is called the **User Interface (UI)** of the app. A user interface is a screen that contains **components** through which someone can interact with an application. As we can see on our camera app, the User Interface contains a button at the bottom centre, and we take pictures by clicking on that button. That is a *Button* **Component**, and it performs the **action** of actually capturing the image using the phone camera. The process of clicking the component of the user interface, in this case the shutter button, causing it to capture the image, is called **interacting with the user interface**. There are some user interface components that can be interacted with, like buttons, while there are some that can only be seen and don't react upon interaction, like text.

Every app you will ever use has this same structure: A User Interface that contains components, some of which perform an action when they are interacted with. You can check out other applications on your Android device and see if this is true of them.

This is something to keep in mind, as you start your App Development journey. The first thing to do when creating an app is to think about all the functions that the app performs. These functions are usually called **features**. Identify the components that are needed for the user to interact with in order to perform those functions. Then, design a User Interface with these components, and finally, add the ability to perform those functions when those components are interacted with.

Let us look at our example, the camera app, again. If long ago, when there were no camera apps, we were tasked to develop an application that can use the camera on an Android device to capture images, how would we go about developing it? As stated above, the first thing would be to determine all the functions the app would perform. In this case of a basic camera app, the only function we want to perform is to **capture an image using a camera**. Now that we have determined that, the next thing is to decide which component we need, to perform that function. We know that is a **Button**. We then design a **User Interface**

and place a Button component inside it. Finally, we make the app capture an image when the button component is **clicked**, i.e. when the button is interacted with.

Below are the processes we will use for every application we will be creating in this textbook. We can easily come back and look through it:

- **Determine** all the functions needed in the application.
- **Decide** which components are needed to perform these functions.
- **Design** a User Interface and arrange the components in it.
- **Program** the components to perform their functions when they are interacted with.

Great! Using these processes, we are going to create many fun apps and games together throughout in this textbook. At the end, you (the student) will be able to create any Android application of your choice. [cool face emoji]

To recap,

- Everything done in an Android device (smartphones and tablets) are done through the use of apps. Nothing can be done without an application.
- An application contains a User Interface which consists of components.
- Some components can be interacted with to perform a specific function or group of functions.

Now that you understand the connection between an Android device and Applications, let us see how we can create our own Android applications.

App Inventor

For all our app development in this textbook, we will be using a platform called App Inventor. App Inventor is a **visual** programming platform that allows you to develop Android apps using visual elements and a drag-and-drop interface. This makes the process of developing apps less tedious and less time-consuming.

App Inventor was developed by an organization called MIT. Later on, you will see it referenced elsewhere as MIT App Inventor.

App Inventor contains all the necessary tools that we need to follow the processes listed earlier for developing apps. It also has everything we need to package our developed apps and send them to an Android device for installation and usage.

Simply imagine that App Inventor is your kitchen. When you want to prepare something to eat, you go to the kitchen, grab your pot, spoon and other kitchen utensils. After some minutes of cooking, your food is ready to be eaten, right? In the same way, App Inventor is our “app kitchen”.

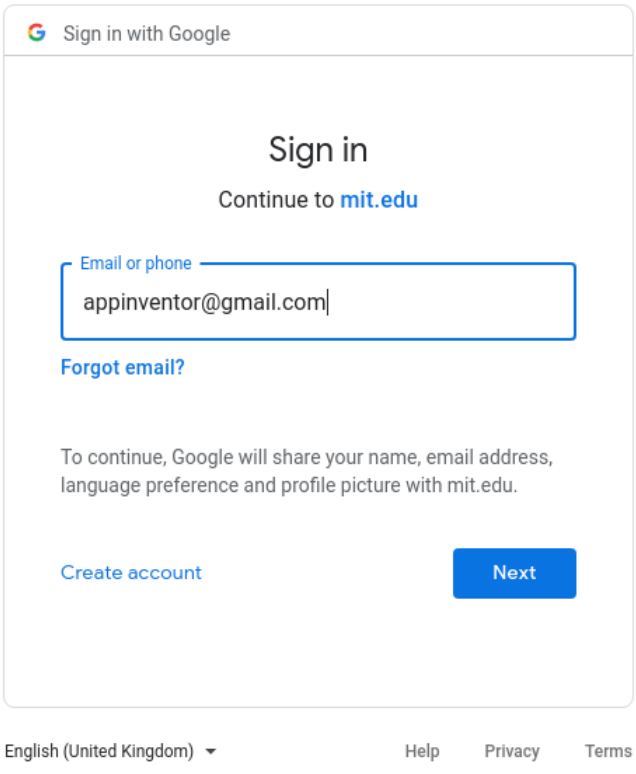
Whenever we want to make apps, we go into App Inventor (kitchen), use the tools (kitchen utensils) available on App Inventor, develop our app (food) until is done and ready for use (to be eaten).

And just as you do not go on eating the food directly from your cooking pot (you dish it out onto a plate and take it to the dining room to start feasting on the delicious meal you have made), you cannot use the apps you build directly on App Inventor. When you are done developing (cooking) the app, you have to build it and send to an Android device (dish onto a plate) where you can then install it and start using the app (feasting on the food). That is it! You now have a basic idea of what MIT App Inventor is, and what we will be using it for throughout in this textbook.

Let's now go ahead to set up our work environment on App Inventor

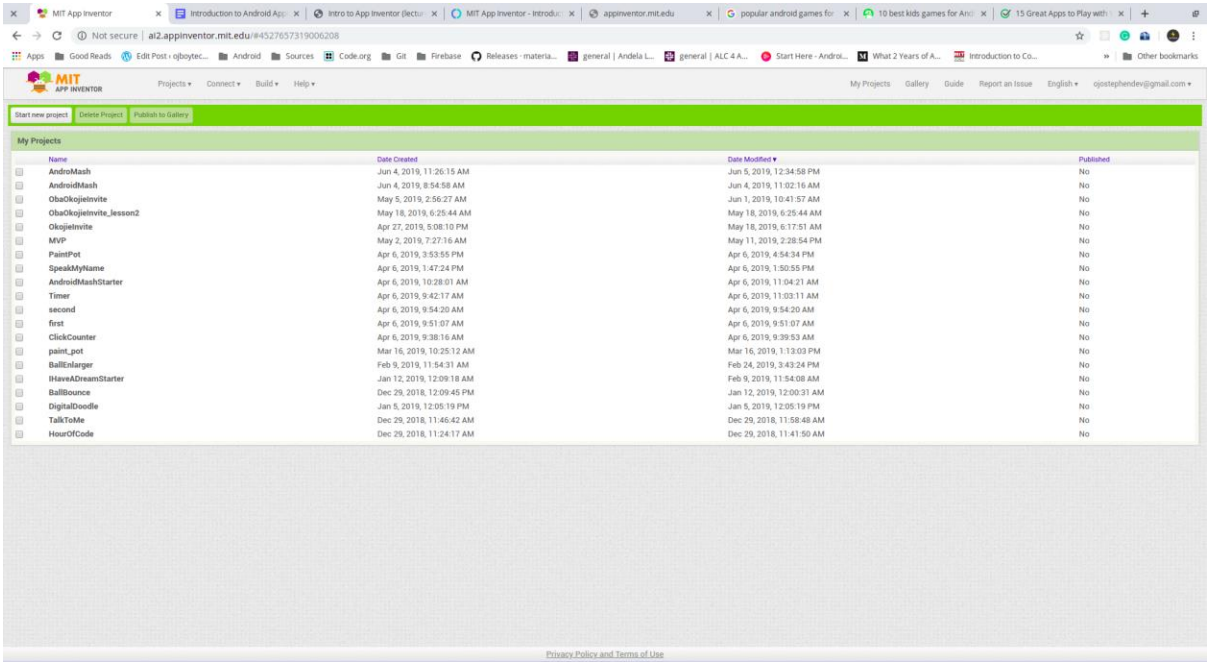
Setting Up Your App Inventor Work Environment

Open a browser on your computer, then go to the web address **ai2.appinventor.mit.edu**. You'll be asked to sign in with a Google account.



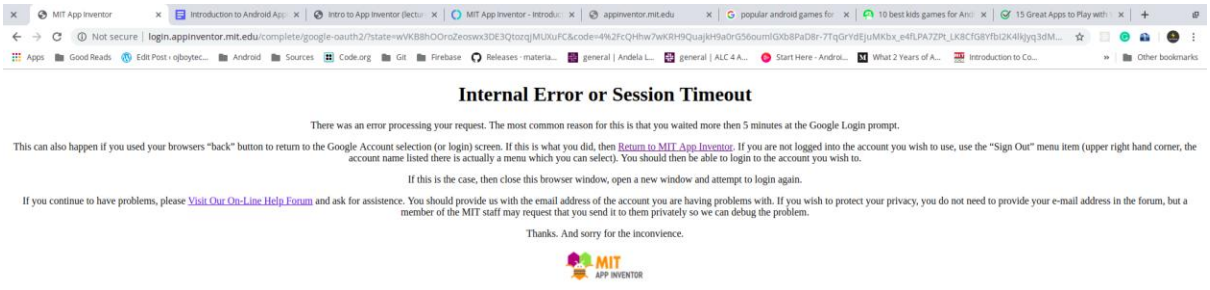
Ask your guardian to help you sign in with a Google account, or if you know how to do it yourself, ask for the sign in details (email address and password) and sign in. Once you do that, you will be redirected to your **App Inventor dashboard**, which is where you can see all the apps you have ever built with App Inventor.

It should look like this:



Since you have not built any app with App Inventor prior to this time, your dashboard will be blank under the “My Projects” section. Don’t worry, we will be filling that space up with a couple of apps very soon.

You may, however, run into errors and you might get a screen that looks like this instead:



Don't panic, just click on the text that says "Return to MIT App Inventor" or load the web address ai2.appinventor.mit.edu again and you'll be asked to sign in again. Do that, and you will be finally taken to the dashboard that looks like the first image above.

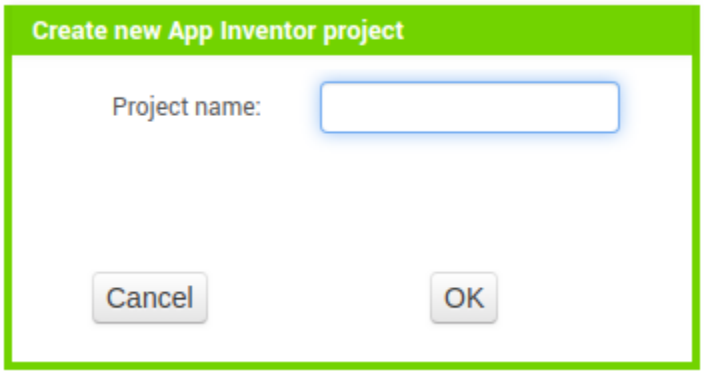
Starting an App Inventor Project

To create an application with App Inventor, you need to create a project for the app. An App Inventor project contains everything you will need to develop and package a single app, so this is how it goes: **An App Inventor project for an app**.

Let's create a sample project so that you will see how it works.

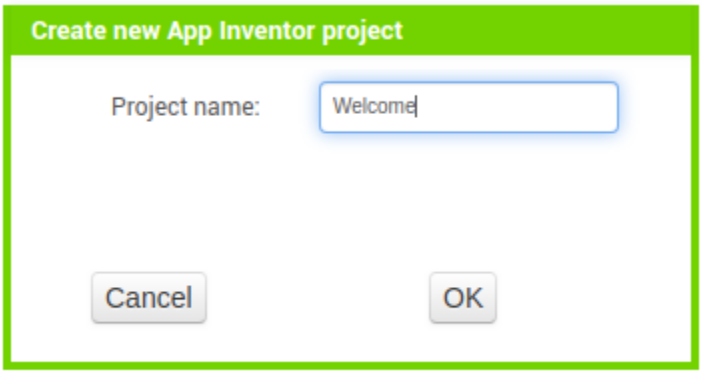
On your App Inventor Dashboard shown earlier, click on the "Start new project" button located in the top left corner of the screen.

You'll be asked to name your project.



You can call your app whatever you want, but I will call mine "Welcome".

NOTE: Your project name cannot contain spaces, so if you wish to have multiple words in your project name, join them together and capitalize the first letter of each word. For example, if you want to name your project "First App", you'll use "FirstApp".



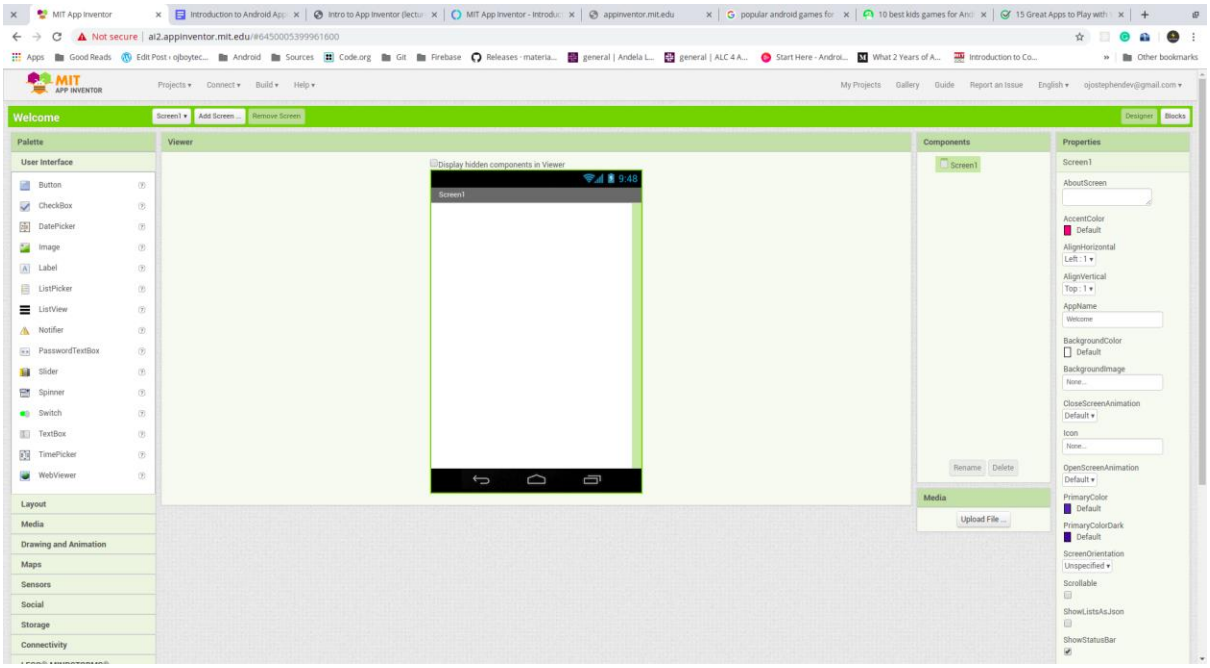
When you have entered your desired app name, click on “OK” and wait for a few seconds; the app development screen for your new app will be opened up. Now, this is our kitchen, set to cook new food.

Introduction to the Designer Window

After creating your project, the first screen that opens up is shown below. The screen is divided into different sections, and the entire window is called the **Designer Window**. This is where we will be doing the first three steps in our process of creating Android apps, which are:

- **Determine** all the functions needed in the application.
- **Decide** which components are needed to perform these functions.
- **Design** a User Interface and arrange the components in it.

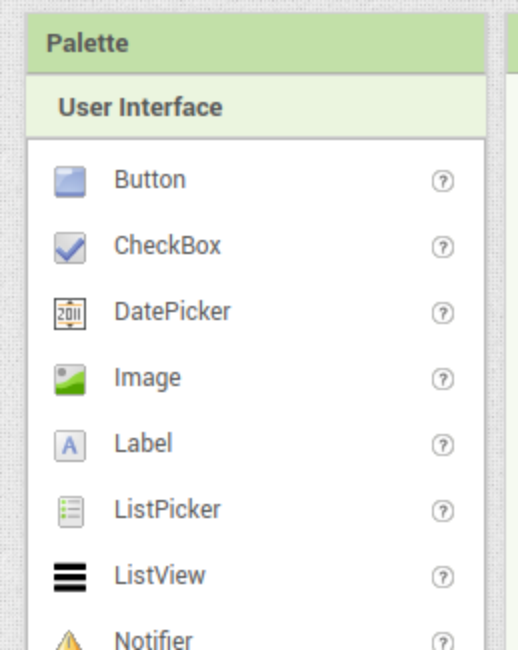
The Designer Window is used for creating User Interfaces with components when developing Android apps with App Inventor.



There are five sections present in the Designer Window. These are:

- Palette
- Viewer
- Components
- Media
- Properties

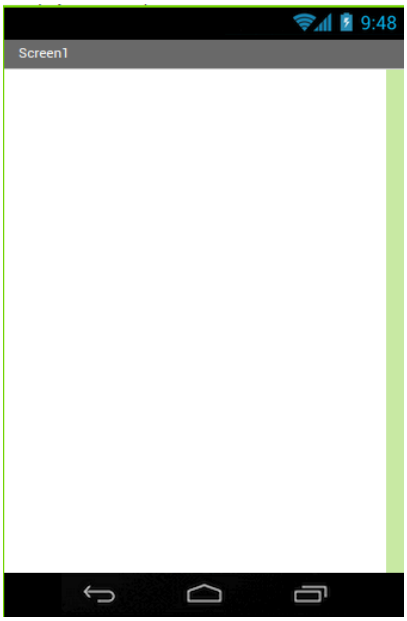
Palette



The Palette is where all the components that we can use for our apps on App Inventor are contained. The components are grouped according to their types; you can see that the first group which is named “User Interface” contains components like Button and TextBox that we can use for our apps’ User Interfaces.

When we determine what functions our apps need and what components are needed to perform those functions, the Palette is where we come to get those components and add them to our apps.

Viewer

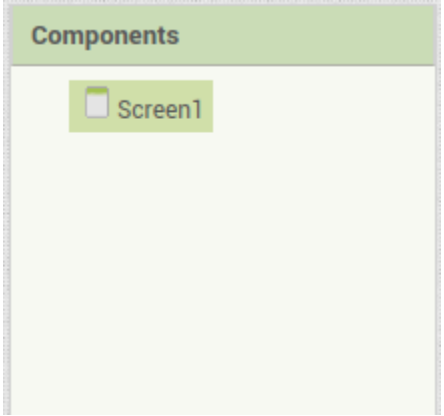


The Viewer contains just one thing: an image that looks like the screen of an Android device. The Viewer is where we arrange the components we'll be using in our app during development, and the way the app screens appear in the Viewer is the way they will appear when you eventually run the app on an Android device.

To add a component to your Android app, you locate the component in the Palette and then drag it to the position you want on the screen present in the Viewer. Components in the "User Interface" group will appear on the screen, while other components will appear below the screen.

You can go ahead and try it out. Drag a component from the User Interface group to the screen, and drag a component from any other group to the screen. Where does each component appear?

Components



The Components section contains a list of all components we have added to the Viewer from the Palette. When you create a new App Inventor project, this section contains only one element, which is the default Screen for every app. Every component you add to the Viewer will appear under the Screen component here.

Go ahead and try it out. Add a component to the Viewer from the Palette and look at the Components section; does it appear under the Screen component?

In this section, you can rename and delete a component from the Viewer. Every component you add to the Viewer is given a default name which is an addition of the general component name and a serial number.

That is, if you add a single Button component to the Viewer from the Palette, the name given to the component will be:

(General Name) Button + (Serial Number) 1 = (Component Name) Button1

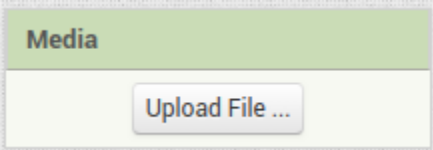
If you add a second Button component, the serial number will be increased and then added to the general name of the component to give the component its own name:

(General Name) Button + (Serial Number) 2 = (Component Name) Button2

This is done so that each component you add to your app has a unique name, a name that is different from the names of every other component in the app.

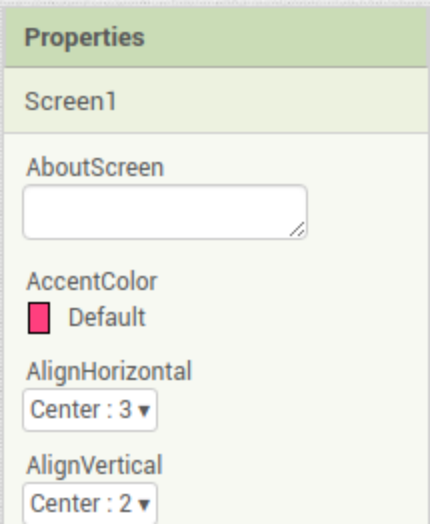
It is a good practice among app developers to give a component a name that best describes what the component does. For example, the button that captures the image when clicked in a camera app can be named “CaptureButton”. This helps you to quickly remember what a particular component does just by looking at its name, and it is useful when you start building complex apps with a lot of components of the same type.

Media



This section is used to upload media files (images, audio or video files) for use in your app. You will learn more about this section and how to use it for app development later in this textbook.

Properties



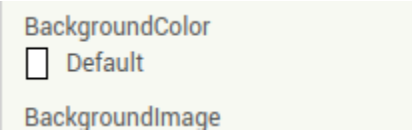
A component has several properties that define its appearance and behaviour. For example, a Button component can display some texts that describe what it does, and the Button can be black or red depending on what its colour property is set to.

The Properties section allows us to view and edit all the properties of a component. The properties displayed in this section are for the currently selected component in the Components section. When a new App Inventor project is created, the Screen is the only component in the Viewer section, hence it is selected in the Components section and its properties are displayed in the Properties section.

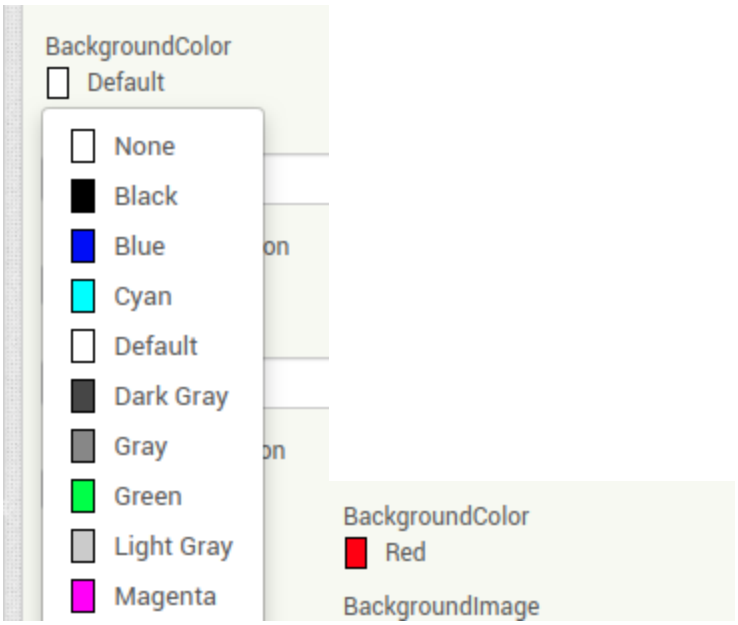
Add a component to the Viewer from the Palette, then select it in the Components section and look at the Properties section; do you see the component's properties? Can you tell what each property means?

A component property has two parts: a **name** and a **value**. The name describes what the property is, while the value describes the current state of the property.

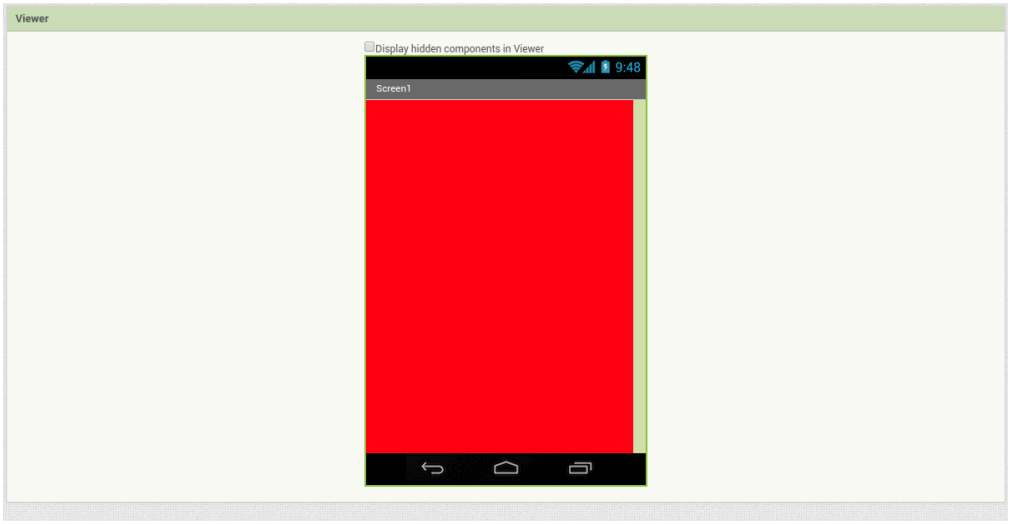
Let us take our Screen for example. Select the Screen component in the Components section and look for a property called "BackgroundColor" in the Properties section. This property describes the background colour of the screen, and its value is "Default" which is White, thereby making the screen's background in the Viewer white.



Now click on the value and change it to another colour.



What happens to the screen in the Viewer section?



The background colour has changed to red, just as you have done in the Properties section.

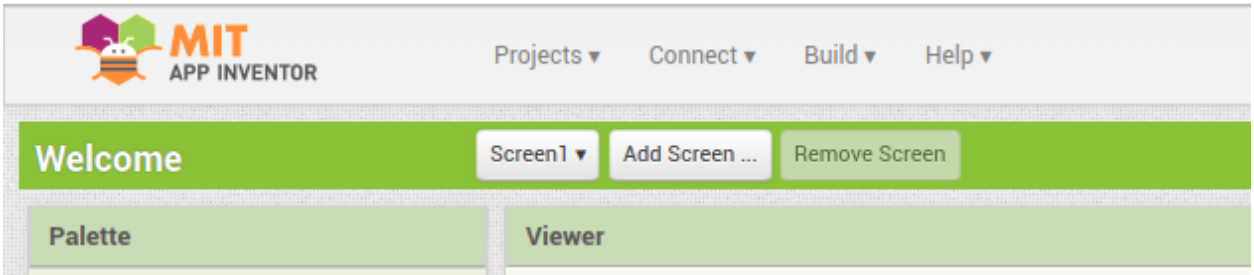
That is all about the Designer Window. Now, let us use what we have learned so far to create your first Android application: “HelloWorld”.

Module Project: “Hello, World” Application

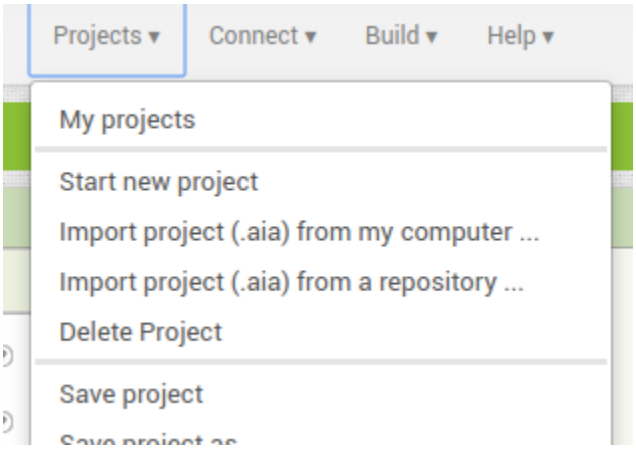
It is customary amongst programmers and application developers to create, as their first-ever application, a “Hello, World!” application. Using our knowledge of the Designer Window on App Inventor, let us create

your first Android application: an app that displays the text “Hello, World” when it is launched. Let us name the app “HelloWorld”.

To start a new project for our HelloWorld app from the current project, click on “Project” at the top left of the screen:



Then click on “Start new project”



Give your project the name “HelloWorld” and you will be taken to the Designer Window for your new app project.

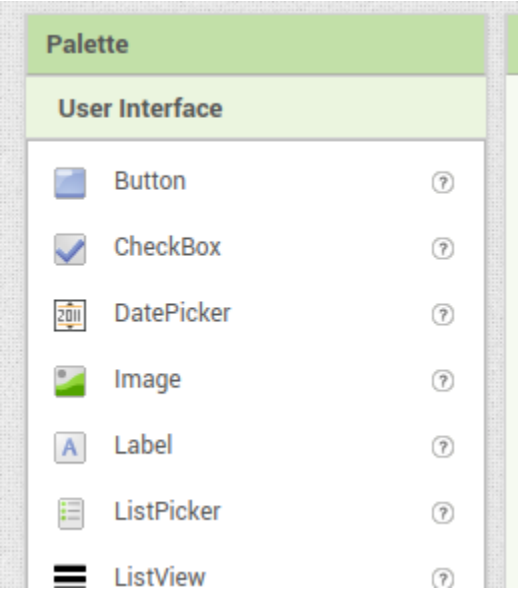
Remember the established processes for developing Android applications? It is time to put it into practice!

Let us follow it step by step.

1. **Determine** all the functions needed in the application.
   Our app has a simple purpose, which is to display the text “Hello World”.

2. **Decide** which components are needed to perform these functions.
   Our app is only displaying text, hence the only component we need is one that can display text. Next, we have to determine what group of components such a component could fall into. A component that displays text would surely be displayed on the app screen, therefore it belongs

in the “User Interface” group. Now let us look through the components in this group and see which one best suits our needs.

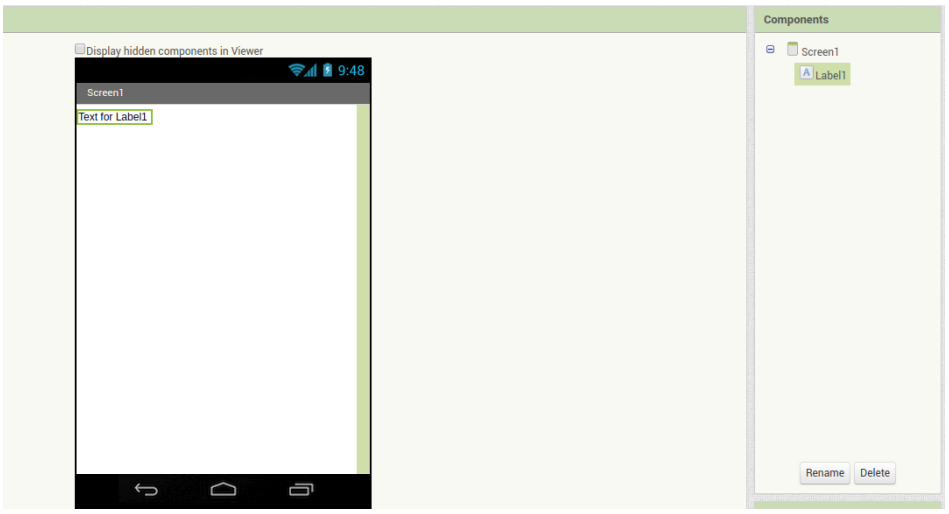


The first component is Button. While a Button can display text, it has additional features like the ability to click, and we do not need this in our HelloWorld app.

CheckBox? No. Next. Definitely not DatePicker. Next one, please.
We do not need anything like an Image. Proceed to the next one.
Label? Sounds interesting. A label is used to describe something and usually contains just text, which sounds perfect for what we want.

Now we have **decided** which component we need for the function of displaying text in our HelloWorld app.

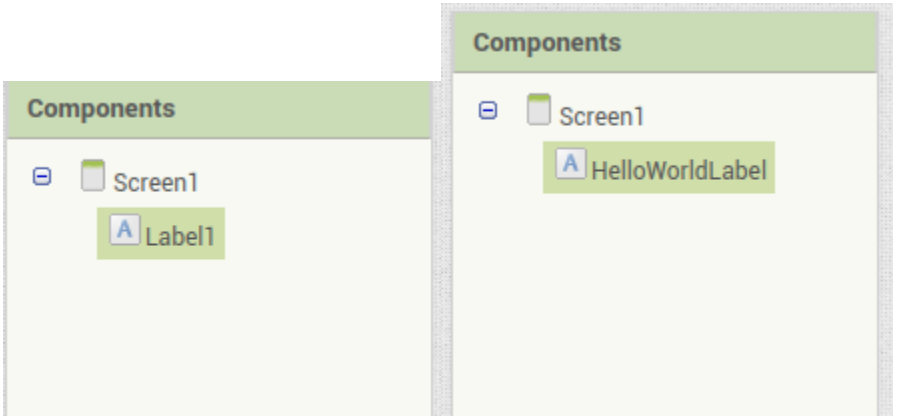
3. **Design** a User Interface and arrange the components in it.
   It is time to actually design the user interface. Drag the Label component from the Palette to the screen in the Viewer and you will see it appear on the screen and in the Components section.



You can see that the Label already contains and displays some text. This is called the **default** text for the label, and it is the initial value of the "Text" property of the Label component. We need to change this to display "Hello, World!".

Before we do that, however, it is important to establish a rule: before doing anything else, always change the name of every new component you add to your app, to a name that describes what the component does.

Our Label displays the text "Hello, World!", so, let us rename our Label1 component to HelloWorldLabel.



Great! Now let us make our Label display what we want: “Hello, World!”.

To do this, we have to look for the property of the Label that describes what we want to change (modify), which in this case is the text. Select the Label in the Components section, and then look through its properties in the Properties section. Which one do you think we can use to change the display text?

I found a property called “Text”, and its value is “Text for Label1”. This is the same text that is currently displayed in the label in the Viewer, so this must be what we need.

Change the value of the Text property to Hello, World! and press the “Enter” key on your keyboard to save the changes. Does the text displayed in the label in the Viewer change too? That is it! We have successfully developed our HelloWorld app.

But we still have one more step left in our app development process...

4. **Program** the components to perform their functions when they are interacted with.

   We do not need any extra functions for our HelloWorld app, so we are going to skip this step for now. We will revisit it later on when we start building apps that require extra functions.

   Now, we have developed an Android application. But we cannot say we have **built** an application yet, because everything we have done so far has been on the App Inventor platform; we have not actually run our app on an Android device yet.

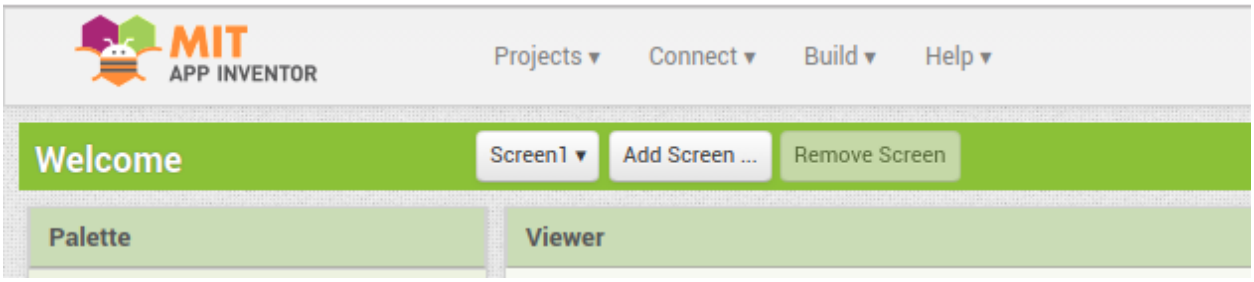
   To be sure that the app you have developed actually works as you expect it to, you need to run it on an Android device or emulator.

Testing Our App on an Android Device

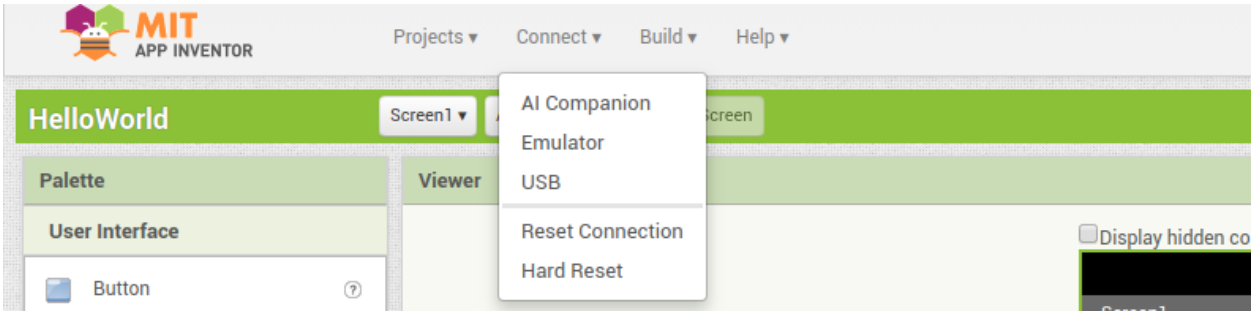
We will be using the AI2 Companion app you installed earlier on your Android device to help us with this process.

These are the steps we need to take to run our app on our Android device:

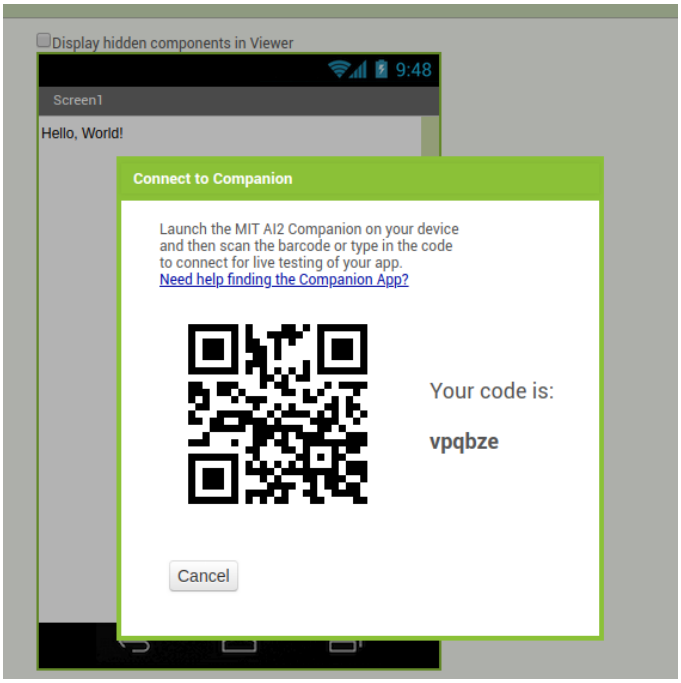
1. Make sure your Android device and your computer are connected to the same WiFi network.
2. Click on "Connect" at the top left of your Designer Window.



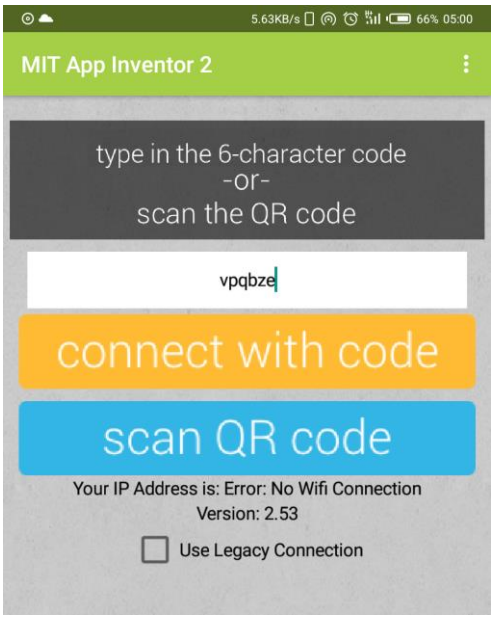
3. Click on AI Companion

4. 

A box will appear with an image (called a barcode) and a six-digit code.

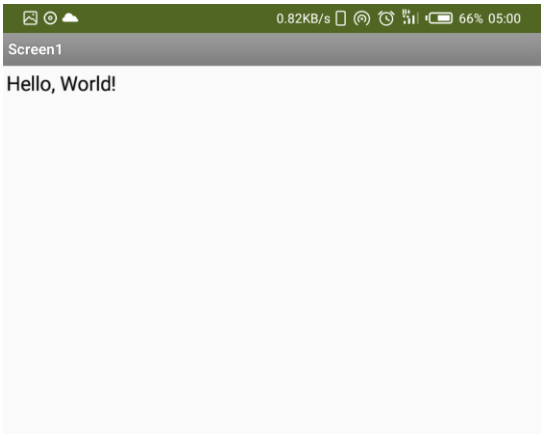
5. 

6. Open the AI2 Companion app on your Android device, type in the six-digit code into the white box and click on "Connect with code".

7. 

   Wait for a few seconds while your Android app gets built and installed on the Android device. When it is done, your app will appear on your Android device just as it is on your App Inventor Viewer.

Does it appear like the one below?



Congratulations! You have just developed and built your first Android application. You can officially call yourself an Android developer!

SUMMARY OF MODULE ONE

In this module, we learned that apps are the main constituents of android devices. These apps are made up of User Interfaces and Behavior, and User Interfaces are made up of components. Then, we learned the processes involved in building an application, and that App Inventor is a visual programming platform that can be used to build Android apps. Also, we learned how to set up an App Inventor work environment and create an app project, how to navigate the Designer Window and create a basic app user interface, and finally, how to run App Inventor projects on an emulator and an Android device.

MODULE TWO

INTRODUCTION TO USER INTERFACE (UI) DESIGN

Module Objectives

At the end of this module, the student should be able to:

- **identify, use and explain the functions of the common UI components in App Inventor.**
- **use the common UI components available in App Inventor to design basic user interfaces for Android apps.**

What Is "User Interface"?

The words **User Interface** is a term you will be coming across quite often as you venture into the world of app development. So, it Is very important that you get the right idea early on. The term is made up of two words: *user* and *interface*. The word *user* describes the end consumers of your apps, that is, the individuals who use your finished Android app.

The word *interface*, according to Merriam-Webster Dictionary, is the place at which independent and often unrelated systems meet and act on or communicate with each other.
Putting these two words together, a **user interface** is the means by which the users of a particular system or mechanism interact with that system/mechanism to utilize its functions. In the context of app development and programming, a user interface is the means by which the *user*, as defined above, interacts with your app and utilizes its functions.

The world around us and the things we interact with in our everyday lives are filled with user interfaces. Human beings, and many other living things that are smart enough to intelligently interact with a system

or mechanism do so with their sense organs, which in the case of humans include the eyes, nose, ear, tongue and skin.

When you are cooking, the food provides a user interface for you to interact with it, which is through its aroma. And you use one of your sense organs, the nose, to carry out that interaction. When you are done cooking the food and it is time to eat it, you interact with the food using another sense organ, the tongue. You also like to see the food you are eating, so you interact with it using your eye.

When it comes to software development, you can interact with applications on Android devices using only your senses of sight (eye), hearing (ear) and touch (skin). Every app has a user interface that you can see, some come with sounds that you can hear, and you have to touch the components of an app through your phone screen to perform different functions offered by the app.
This means that Android apps can contain components that:

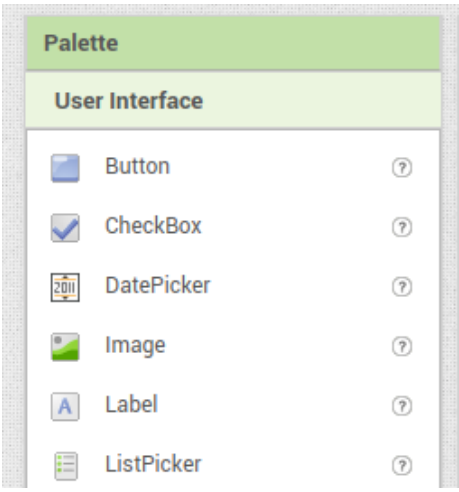
- display material for the users to see, e.g. text and images (the eyes).
- play material for the users to hear, e.g. audio files and sounds (the ears).
- play material for the users to see and hear, e.g. video files (the eyes & ears).
- display material for the users to interact with by clicking, e.g. buttons and links (the skin).

The term *User Interface* is commonly abbreviated to **UI**.

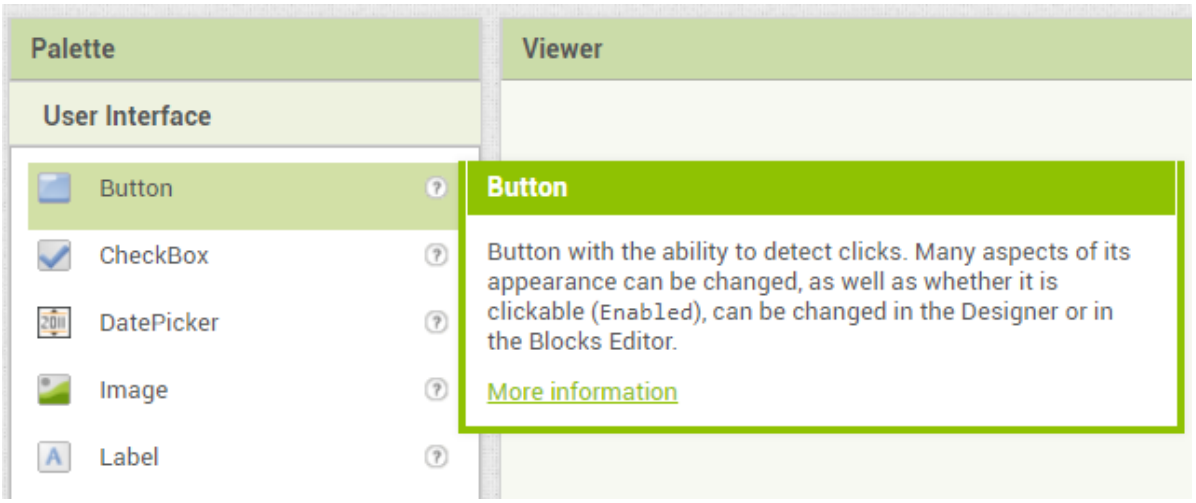
Exploring Built-In User Interface (UI) Components and Their Properties in App Inventor

We learned in the previous module that a User Interface is comprised of components. App Inventor comes with a number of components that can be used to design a *user interface* for our apps. In the last module, we were briefly introduced to these components which are contained in the **User Interface** group, under the Palette section which is in the Designer Window.

There are 15 User Interface components available in App Inventor.



To see what a component is used for, click on the ***'?'*** icon at the right side of the component in the palette.



TRY IT OUT: Go ahead and check the description of each UI component in the palette by clicking on the '***?'*** Icon, to understand what they are used for.

Now let us look at the properties of each component and how they can be changed to modify the component's looks and behaviour.

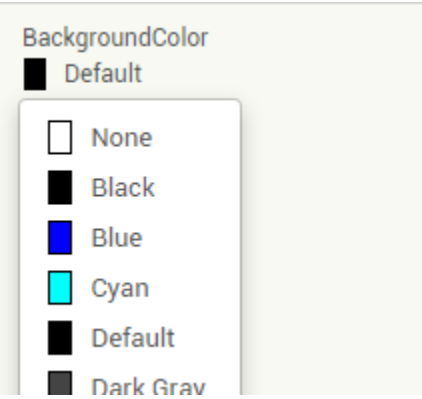
How to View and Modify Component Properties in the Properties section

To view a component's properties, you have to add it to the Viewer. The components in the Palette represent the **types** of components that can be used in App Inventor, and dragging one from the Palette to the Viewer adds a component of that type to your application. All components of a type have the same

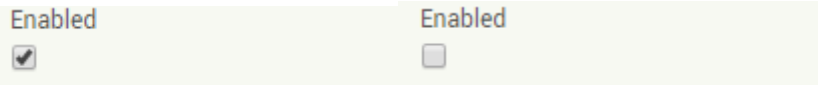
initial properties, but the properties of individual components of a type can be modified without affecting other components of the same type.

This can be likened to cars, for instance. One of the numerous car types available in the world today is Toyota Corolla. There are hundreds of thousands of Toyota Corolla cars all around the world, but they are not all exactly the same (in terms of looks or behaviour) at this moment. They all had the same properties when they were produced by the manufacturer, but then the colour property of some were changed to blue, some to red, some white and so on. Also, when individual units of these cars are bought, the owners might decide to change the interior design of these cars, or what is printed on the body of the car. Mr A changing the colour of his Toyota Corolla to blue does not change the colour of Mr B's red Toyota Corolla to blue.

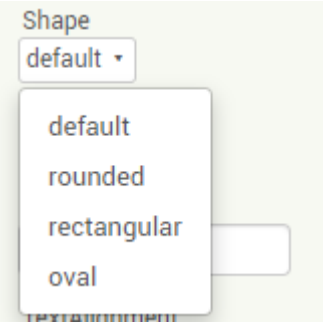
Structure of a Component Property

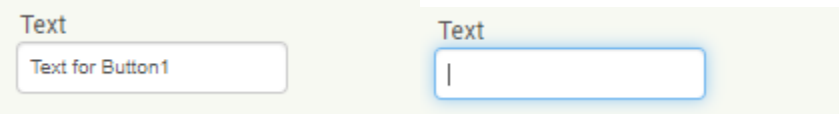


[This picture should be modified to indicate the structure of a property as follows: the "BackgroundColor" text is the Label, the "Default" section is the Current Value, and the expanded section is the Value List - select a value to modify the property.]



[This picture should be modified to indicate the structure of a property as follows: the "Enabled" text is the Label, the checkbox section is both the Current Value and the Value Modifier; check and uncheck the box to modify the property.]





All component properties have two sections: A **Label** which describes the property, and a **Value Modifier** which shows the current value of the property, and can be used to modify the value of the property. There are four different types of Value Modifiers as shown in the images above.

Most UI components in App Inventor display some texts that can be used to describe the component's function(s). Hence, most components have some properties in common.

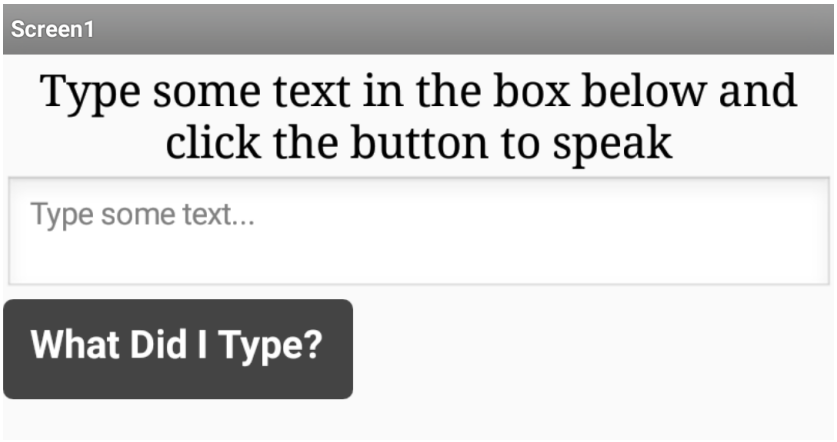
TRY IT OUT: For each component, type under the User Interface group in the palette, add a component to the Viewer and look at each of its properties under the Properties section. Can you guess what each property describes from the Labels? Change the values of each property and observe how the looks and behaviour of the component change (use your emulator or an Android device to get a real-life demonstration of this).

Module Project: *TextToSpeech* App User Interface

For this module, we are going to build an app that allows the user to enter text and then, click on a button to speak the text out. Since we only just learned about User Interfaces, we will build the UI of the TextToSpeech app in this module, and complete it when we learn about adding functionalities to apps in the next module.

This is the User Interface we intend to achieve:

In the Emulator



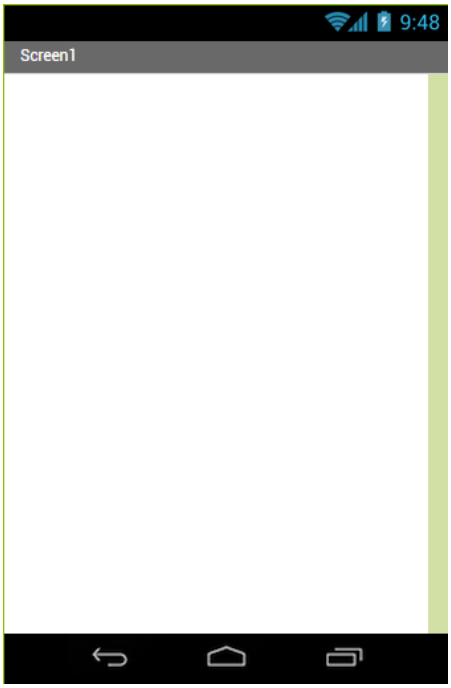
In an Android smartphone

Remember our established process for developing Android applications from the last module? Let US put it into practice once again.

The first step is to **determine all the functions needed in the application**, but since we are building just the UI for this module, let us jump to the second step which is to **decide all the components needed in the application**.
From the final app images above, we need three types of components in our application:

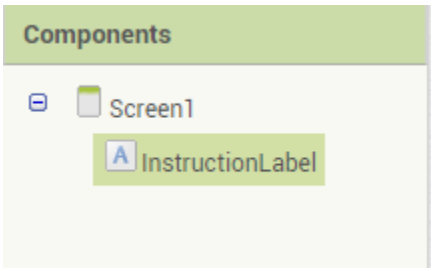
- A component that displays text. The UI component that best suits this function is the **Label**.
- A component that allows the user to enter some text. If you attempted the earlier "Try it out" section and went through the description of all the components under the User Interface group, you would know it is the **TextBox** component that best suits this purpose.
- A component that the user can click on to perform some functions. The component we need here is most definitely a **Button**.

Now that we have identified the components we need in our app, let us start adding them to the Viewer and modify their properties so as to arrive at the final UI in the images above.



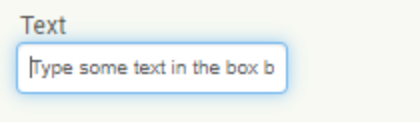
Starting with a clean slate.

Drag the **Label** component from the Palette to the Viewer
The first thing to do when adding a component to your app is to change its name to something that describes what that component does. Since this label is used to give an instruction to the user, let us rename it **InstructionLabel**.



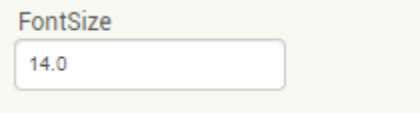
With that done, let us change the text displayed by the label to our instruction. To do this, we have to look for a property of the **InstructionLabel** component that we can use to modify the text. Look through the component's properties; which do you think it is?

The **Text** property has the value "Text for Label1", which is the same text displayed on the label, so, this will be more appropriate. Change the value to "Type some text in the box below and click the button to speak".



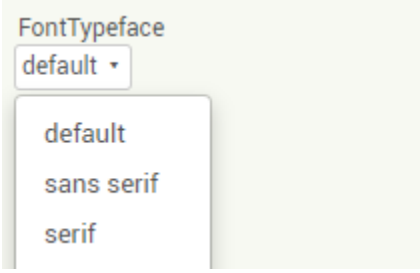
Your app should now look like this:

The text displayed is correct, but it looks a lot smaller than the text in the final version. Let us look for a property that we can use to change the size of the InstructionLabel.



Increase the value of the FontSize property and observe how the Label gets bigger. The size of the Label in the final version is 20.

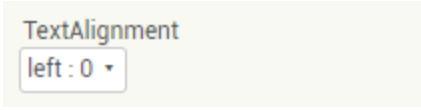
What other difference do we have between our **InstructionLabel** as it is, and the one in the final version? If you look closely, you will notice that the styles of the two texts are different. Can you find the property that can be modified to achieve the text style we want?



Which of these values gives us our desired text style?

There is one more thing left to do for our **InstructionLabel** to look like that of the final version. Notice that the text displayed on the Label is aligned to the left, but that of the final version is aligned to the centre.

Look through the **InstructionLabel**'s properties; which of them can be used to change its text alignment?

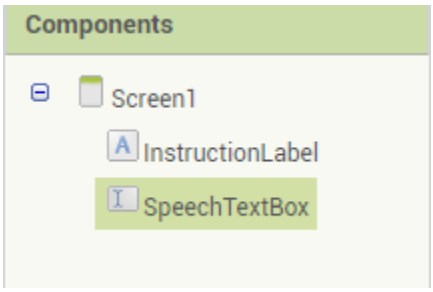


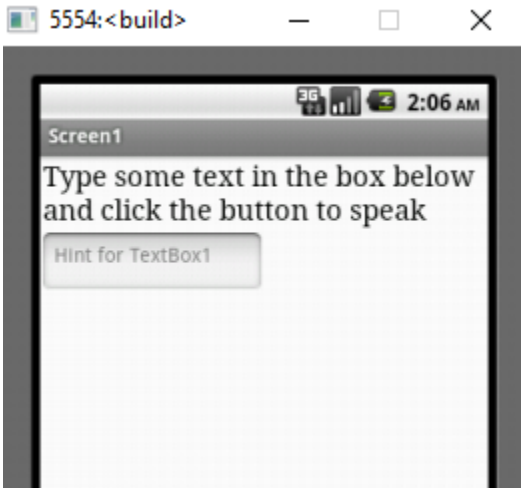
Got it!

You can see that the current value of the *TextAlignment* property is "left", hence the text is aligned to the left. Change the value to *center*.

Our **InstructionLabel** now looks like that in the final version. Great! Let us move on to the next component: the TextBox.

Add a **TextBox** component to the Viewer from the User Interface palette. Then, rename the TextBox to **SpeechTextBox** since the user is supposed to type in the text he/she wants the app to speak into the text box.

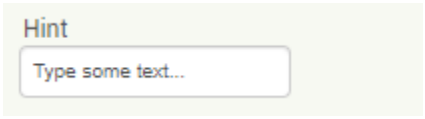




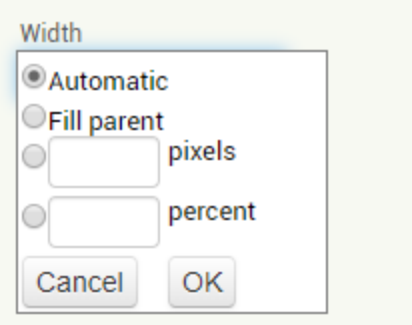
You will see that the TextBox displays a text, “Hint for TextBox1”, but its *Text* property does not have a value. A hint is the text that gives the user an idea of what to type into the TextBox. When the user starts typing into the TextBox, this hint vanishes from the TextBox. The hint shown on the TextBox can be modified with the *Hint* property. This is similar to the *Text* property of a Label component that describes the function of the Label.

Note: If you specify a value for the *Text* property of a TextBox, the text would be entered into the text box and would not vanish when the user starts typing.

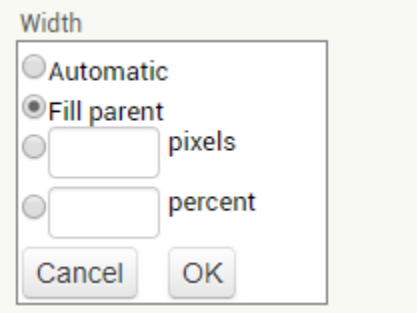
Let us change the hint shown on the SpeechTextBox to “Type some text…”

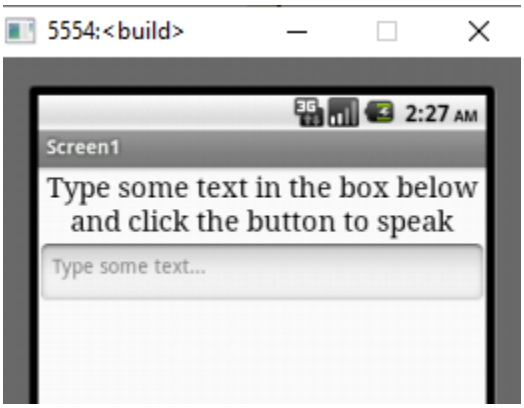


This now looks perfect, but there is one more thing: the SpeechTextBox in the final version expands to fill the width of the device screen, but ours does not. Can you think of a property that can help us fix this?



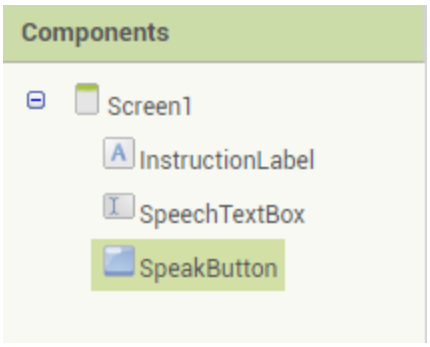
If you are thinking of the *Width* property, you are right! The default value of this property is "automatic", which means the component would only be as wide as the components contained inside it. In this case, the only thing contained inside the SpeechTextBox is the hint text "Type some text…", so, the TextBox is only as wide as the length of that text, since the value of its *Width* property is *automatic*. To make it expand across the full length of the device screen, change the value of the property to *fill parent*.

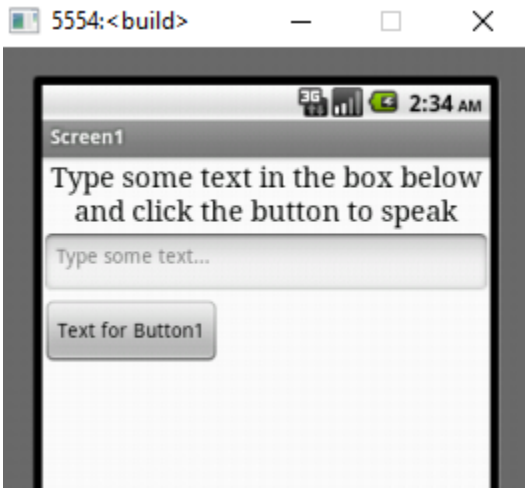




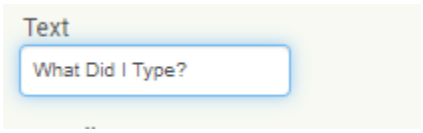
Now, this looks like what we have in the final version. Let us move on to the next and last component: The **Button**.

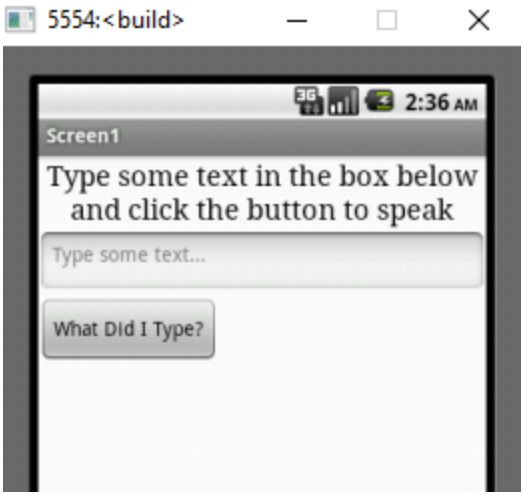
Drag the Button component from the User Interface palette to the Viewer to add it to our app. Next, rename the Button component to **SpeakButton**, since the Button speaks the text entered by the user in the **SpeechTextBox** when it is clicked.



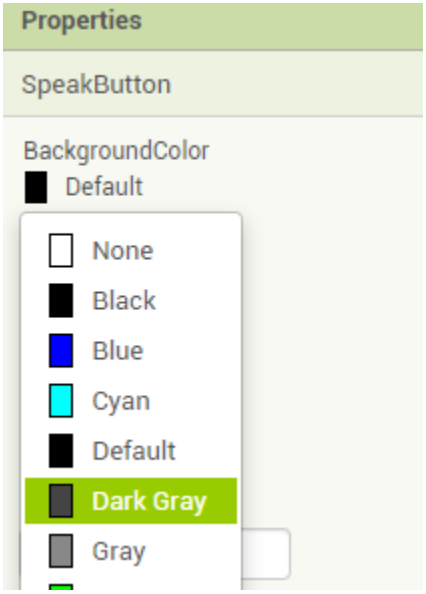


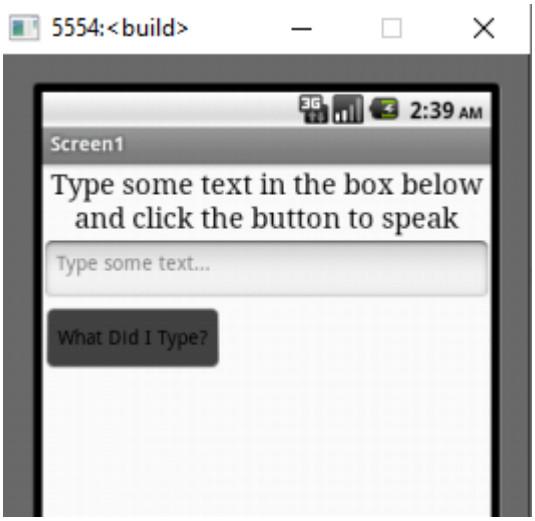
The Button in the final version displays the text “What Did I Type?”. Let’s modify the *Text* property of our SpeakButton to reflect this.



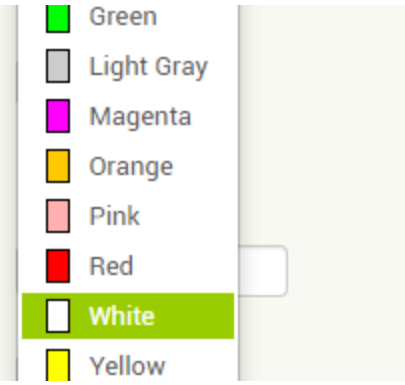


Notice that the colour of the Button (that is, the *BackgroundColor*) in the final version is dark gray. Let us modify the *BackgroundColor* property of our SpeakButton to reflect this.

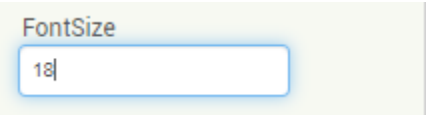


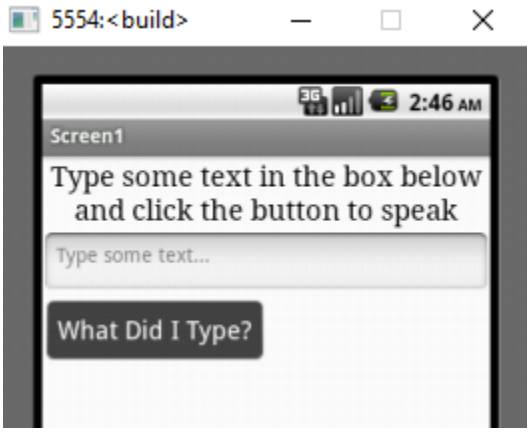


Uh-oh! The text on our SpeakButton is now very difficult to read. This is because both the text and the background on which it is displayed are dark-coloured. A useful rule of User Interface design is to put light-coloured elements on a dark-coloured background, and vice versa. Let us change the colour of the text displayed on the SpeakButton component to white to make it more legible.

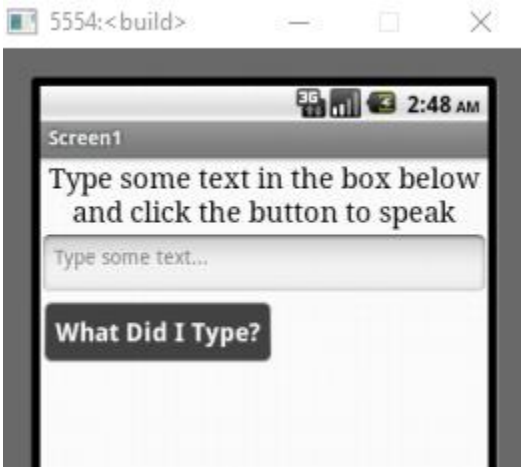


We are getting closer to the final version. The text on the button in the final version looks a bit bigger, so let us try increasing the size of the SpeakButton's text to 18 and see what happens. For this, we are going to modify the *FontSize* property.





Nice! That size looks perfect, but the text in the final version's button looks **bolder** than the one on our SpeakButton. Let us fix that with the *FontBold* property.



This looks exactly like the final version shown at the beginning of the project. Try interacting with your app on the emulator or on an Android device.

- Can you type text into the SpeechTextBox?
- What happens when you click on the SpeakButton?
- Does the InstructionLabel react to click?

We have now designed the User Interface of our TextToSpeech application. However, nothing happens when the SpeakButton is clicked. This is because we have not programmed our application to react to actions like clicks, i.e. we only added looks, and not behaviours, to our app. We will learn how to do this in the next module.

Summary of Module Two

In this module, we learned about what a User Interface is, and its importance in an app. Also, we learned that components have properties that can be used to modify their looks and behaviour. Finally, we learned how to view and modify component properties on App Inventor, and how to design User Interfaces for applications with components.

MODULE THREE
ADDING BEHAVIOUR (ACTIONS AND) EVENTS TO ANDROID APPS

Module Objectives

At the end of the module, the students should be able to;

- **differentiate between App Behaviour and UI.**
- **explain how both are integral parts of an application.**
- **add behaviour and interactivity to Android apps.**
- **use the *TextToSpeech* component to convert text to speech in Android apps.**

What is App Behaviour?

In previous modules, we have established that every Android application we will create would have a User Interface which is composed of components, some of which perform an action when they are interacted with. The User Interface is the medium through which an app can be interacted with by the user to perform certain actions as defined by the app developer. These interactions made with the components of an app's UI- that trigger the app to perform actions, are called **events**- and everything the app does in reactions to these events, that is the **actions**, constitutes the app's **behaviour**. App Behaviour is two-sided: **events** signal an app to exhibit a particular behaviour, and **actions** are used to execute the behaviour.

Let us go back to our camera application. When the user wants to capture an image, they have to trigger the camera app into doing so by interacting with the shutter button, that is, by clicking on it. This act of clicking on the shutter button to trigger a reaction is called a **click event**, and the camera app's behaviour when it receives a click event on its shutter button, is to capture an image with the camera. The User Interface provides the shutter button component for the user to interact with, and the user triggers the app to perform the image-capture function by initiating a click event on the shutter button.

From the above example, we see that an application's UI is connected to its behaviour through **events**. Therefore, an application can be wholly described as software that provides a user interface and exhibits certain behaviour in reaction to events. Without a UI, an app's behaviour, and by extension its functions, cannot be accessed and is thereby useless to the user, and without behaviour, an app is only useful for displaying information to the user.

To recap,

- A fully-functional application is made up of User Interface and Behaviour.
- An application's User Interface is made up of components.
- An application's Behaviour is made up of actions that are performed in reaction to events.
- A user can trigger an app behaviour by initiating an event on the components in its User Interface.

It is not a must that the user interacts with the components on an app's user interface before the app can perform actions. Apps are capable of performing actions without prior user interaction.

Let us take, for example, an alarm application. To set an alarm, the user initially interacts with the components on the app's user interface to set a date and time (and sometimes a message) for the alarm to go off. An Android device broadcasts an event (let us call this a **time event**) every second to let every application on the device know the current date and time. Therefore, the alarm app is programmed to listen to that event, and ring an alarm if the current date and time is equal to the date and time set by the user. When the time comes for the alarm set by the user to ring, the user does not have to interact with the app's user interface; the app is triggered to ring the alarm (that is, to play an alarm tone and vibrate the phone or display a message, depending on the user's settings) by the **time event**. This type of event is called an *external event*.

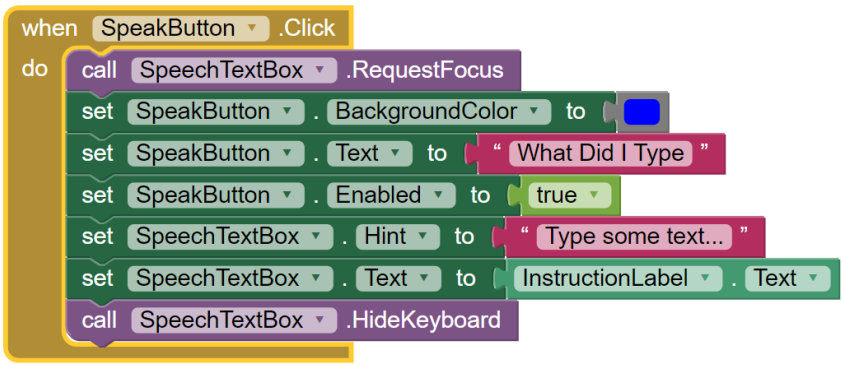
There are other types of events that can trigger app behaviour, apart from user-initiated and external events. As we build more apps in this textbook using App Inventor, we will come across these event types and see how we can program our apps to react to these events.

Programming App Behaviour with Blocks in App Inventor

We have learned that the process of arranging components together on a screen to form a User Interface is called **designing** the app. Likewise, the process of specifying how an app reacts to events to perform actions, that is, specifying its behaviour, is called **programming** the app. To program an application, you need to create codes that the computing device (on which your application would be run on - in this case, an Android device) can understand and execute.

In traditional programming languages, code is written using a set of language-specific keywords and characters. In App Inventor, however, **Blocks** are used to write code to program an application's behaviour. This is what a code block in App Inventor looks like:

More often than not, you would need multiple blocks to fully define a certain behaviour for your app. App Inventor's code blocks can be stacked together, and they will be executed in the order in which they are stacked.



When several blocks are stacked together in this manner, it is called a **Script**. A script represents a particular behaviour of an app.

The structure of a block determines the way in which it can be stacked with other blocks. Some blocks can contain other blocks within them, some can be stacked only above or below other blocks, while some can only be attached as inputs to other blocks (these are called **input parameters**).

Blocks are colour-coded, and the colour of a block can be used to quickly identify what type of block it is and how it can be used in relation to other blocks to program an app. Blocks also display a label which describes their function and what components they are acting on.

Blocks in App Inventor can be grouped into various types, based on their **mode of behaviour** and **mode of operation**. The label, structure and colour of a block depend on its mode of behaviour and mode of operation.

Mode of Behaviour

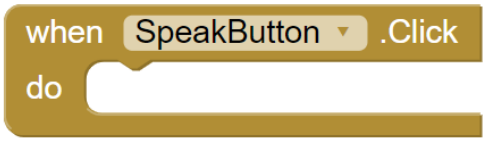
The structure of a block is defined by its mode of behaviour. The types of code blocks that can be used in App Inventor, according to their mode of behaviour, are as follows:

- Event Blocks
- Control Blocks
- Command Blocks
- Reporter Blocks

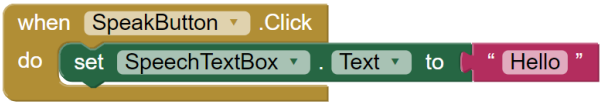
Event Blocks

Event Blocks are used to program an app to react to certain events. We learned earlier that an app reacts to events to perform its functions. Hence, event blocks are used to start a particular behaviour of an app in response to a certain event, that is, a script.

Event blocks can only contain other blocks within them; they cannot be stacked below or above other blocks. All event blocks look like this:



The block above executes (in other words, "**does"**) the blocks and/or scripts in the "do" section **when** a **click** event is triggered on the component named **SpeakButton**.

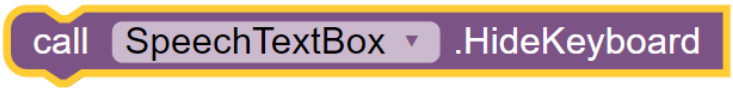


The block above can be read as follows: when SpeakButton is clicked, set SpeechTextBox's text to "Hello".

Control Blocks

Control blocks are used to alter the flow of execution of a script. We will learn more about this type of blocks later in this textbook. A control block can be stacked above or below other blocks, and can contain other blocks within it as well.

Command Blocks



Command blocks are used to instruct the app to perform an action. The block above can be read as follows: tell SpeechTextBox to hide its keyboard. Command blocks can be stacked above or below other blocks.

Reporter Blocks

Reporter blocks are used to report a piece of information or value and hence, are always used as input parameters to the other types of blocks. The block above reports the background colour of SpeechTextBox, and can be used as follows:

The "set SpeakButton BackgroundColor to" block is a **Command block**, and its single input parameter is the "SpeechTextBox BackgroundColor" **reporter block**. This script sets the background colour of the SpeakButton component to the background colour of the SpeechTextBox component

Reporter blocks can also be used to perform operations, but in such cases, the result of their operations are always reported to the script in which they are stacked. For example, the block below subtracts the numbers 10 and 5 and reports the result of the subtraction.

This block can then be used as an input parameter to another block as follows:

Mode of Operation

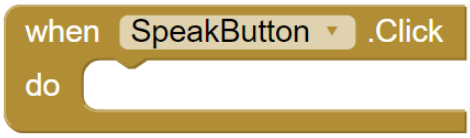
The types of code blocks that can be used in App Inventor, according to their mode of operation, are as follows:

- Component Blocks
- Built-in Blocks

Component Blocks

Component Blocks are used to program components that can be found in the Palette section of the Designer Window. Each type of component has its own set of blocks that can be used to program it. All components can have event, command and reporter blocks.

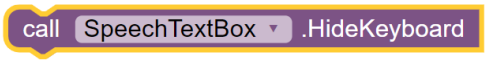
The event blocks that are available for a component describe the events that can be triggered on that component. Component event blocks are usually brown in colour.



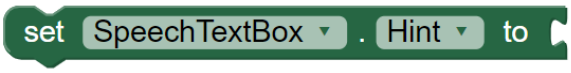
Event blocks are also called **event handlers**, since they are used to handle an app's reaction to events.

The command blocks that are available for a component describe the actions that can be performed on or by that component. There are two types of component command blocks: **action** blocks and **property setter** blocks.

Component action blocks are used to perform an action on or with the component. They are usually purple in colour.



Component property setter blocks are used to set the values of a component's property. They are usually green in colour.



The reporter blocks that are available for a component are used to report the value of a component's property, and are thereby called **property getter** blocks. Like property setter blocks, they are green in colour.

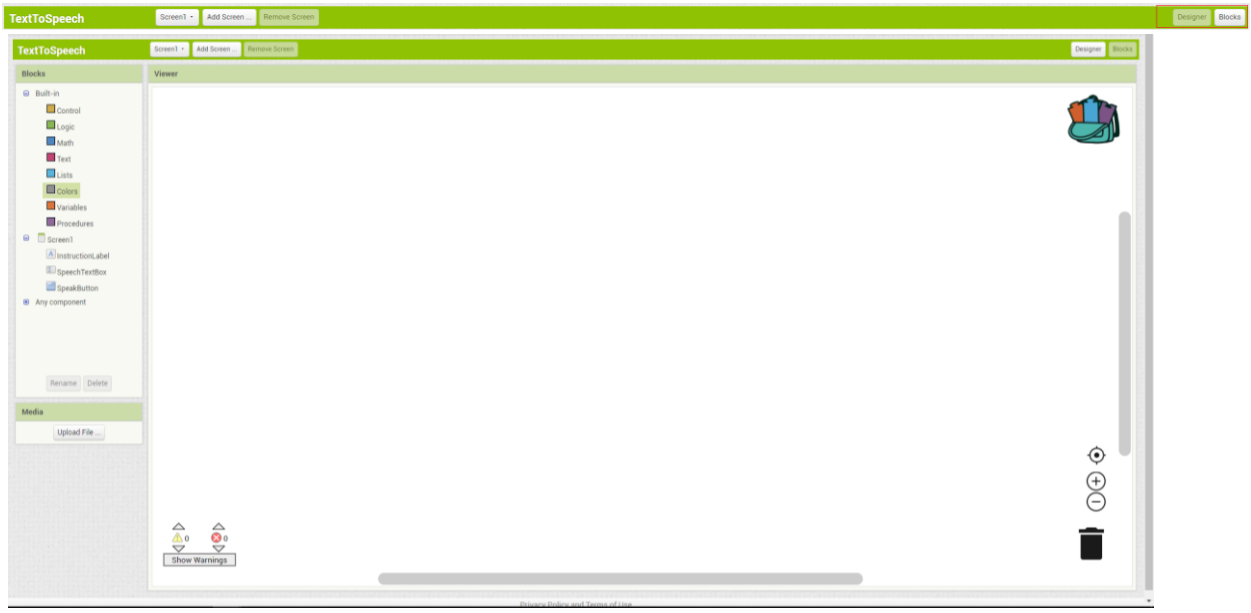
Built-in Blocks

Built-in blocks are used to program parts of an application that are not specific to and do not depend on a component or a type of component. For example, you do not need a component to perform a math operation. There are eight categories of built-in blocks in App Inventor.

To write code using Blocks in App Inventor, you have to switch to the **Blocks Editor**.

Introduction to the Blocks Editor

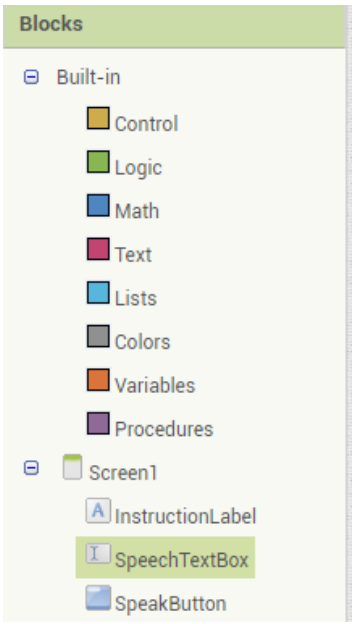
The default screen on App Inventor is the Designer Window, which we use to add components to our apps and design its User Interface. To switch to the Blocks Editor, click on the "Blocks" button at the top right corner of the App Inventor screen.



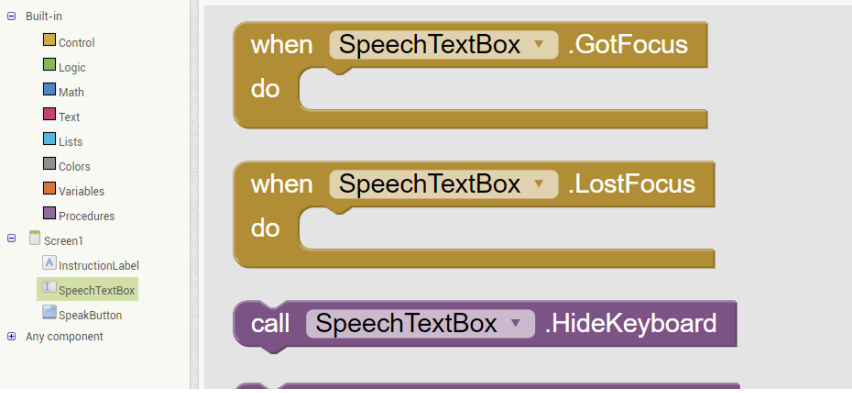
The Blocks Editor has two main sections: **Blocks** and **Viewer**.

Blocks

The Blocks section is where all the blocks that can be used to program applications on App Inventor are contained.



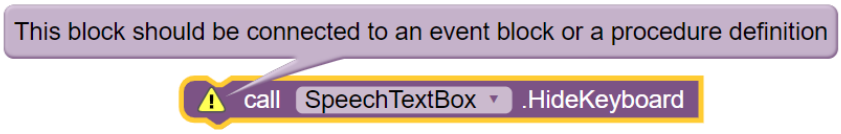
The Blocks section is divided into built-in and component blocks. To view the blocks for a particular component or category, click on the component in the Blocks section.



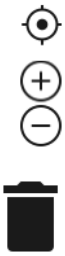
Viewer

Similar to the section with the same name in the Designer Window, the Viewer is where blocks are stacked together into scripts to form app behaviour in App Inventor. To define app behaviour, blocks are dragged from the Blocks section to the Viewer.

The bottom left side of the Viewer shows warnings and errors in your code blocks. For example, if you add a command block outside an event block, you will receive a warning as shown in the image below:



The bottom right side of the Viewer contains a bin. Drag blocks and scripts to the bin to remove them from the Viewer.



The two buttons immediately above the bin are used to zoom in and out of the Viewer, while the third one is used to return to the centre of the Viewer.

The backpack at the top right of the Viewer is a warehouse for blocks and scripts. The blocks and scripts you add from the Viewer to the Backpack would be available for use in every App Inventor project you create with your account.

Now that we have seen the various ways in which we can program an app's behaviour, let us program the TextToSpeech app we started building in the last module to react to click events and speak text entered by the user.

Module Project: TextToSpeech App (continued)

In the last module, we designed the user interface for an app that allows the user to enter some text, and then speaks the text out loud back to the user. This app is useful if you have a long body of text to go through and prefer that it be read to you, rather than you reading it; simply copy the text and paste it into the app, and click the "What Did I Type" button to read it out.

We have completed steps 1-3 of our app development process:

- **Determine** all the functions needed in the application.
- **Decide** which components are needed to perform these functions.
- **Design** a User Interface and arrange the components in it.

Let us continue from where we left off and complete the last step of the process:

- **Program** the components to perform their functions when they are interacted with.

This seems easy enough. We need to add a single behaviour to this app:

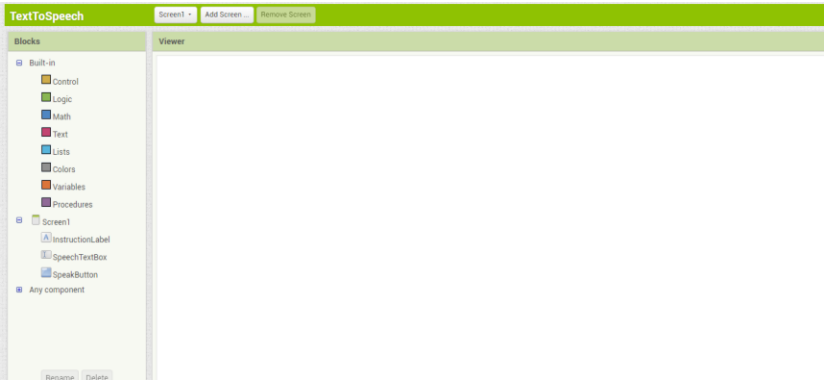
speak the text in SpeechTextBox when SpeakButton is clicked.

As noted earlier, app behaviour consists of events and actions. Therefore, we need to break the behaviour above into events and actions. If one of these is missing, then the behaviour is invalid.

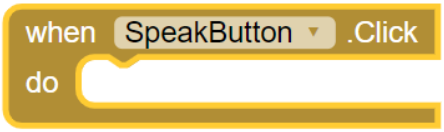
- **When SpeakButton is clicked** - this is a **click event** that is triggered on the SpeakButton component.
- **Speak the text in SpeechTextBox** - this is an **action** that is performed on the SpeechTextBox component in response to the click event.

We have both an event and an action, therefore the behaviour is valid. Let us go ahead and program this into our TextToSpeech app.

Open up your app project and switch to the Blocks Editor.



The first thing to do is to add the event block for the event defined above. The event is triggered on the SpeakButton component, so click on the SpeakButton component in the Blocks section, find an event block that can be used to handle a click event, and drag it to the Viewer.



We have now defined an event handler for a click event triggered on the SpeakButton component. The next step is to add a block or script to perform the action required in our app.
Let us take a closer look at the action:

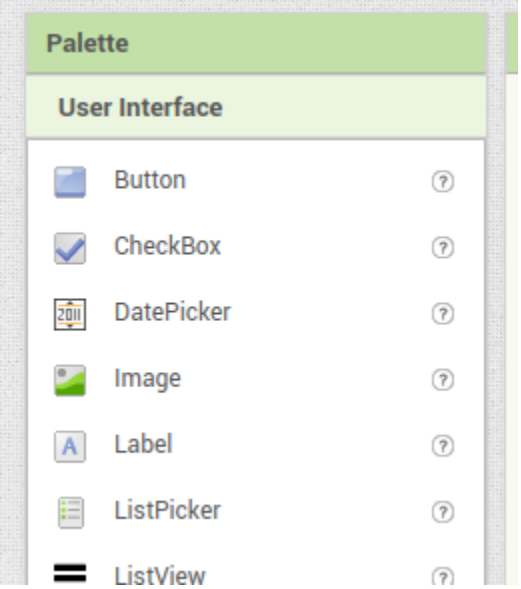
- **Speak the text in SpeechTextBox**

Before we can program our app to speak the text in the SpeechTextBox component, we first need to retrieve it. As the popular saying goes: "*you can't give what you don't have*".

Remember that *Text* is a property of the TextBox component. Therefore, we can retrieve the *Text* property of SpeechTextBox using the **property getter** blocks we have learned about early on

This block reports the value of the Text property of SpeechTextBox. It shows a warning because it is a reporter block and needs to be passed as input to another type of block. We will fix this shortly.

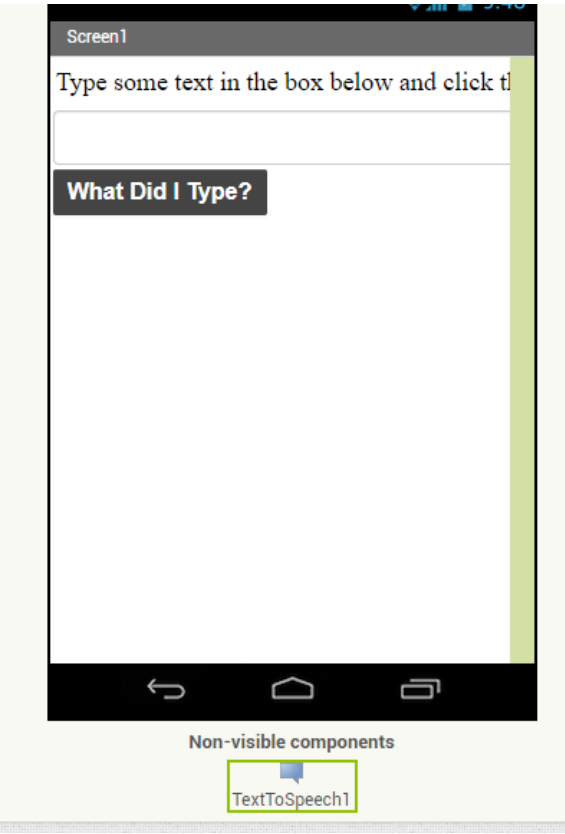
Now that we have retrieved the text in SpeechTextBox, what is left is to program the app to speak this text. However, the three components (Label, TextBox and Button) that are currently in our application are not capable of speaking text. We need a component that is capable of performing this function. Let us go back to the Palette in the Designer Window.



Consider the action we want to perform; it does not have anything to do with the user interface, so it cannot be under the User Interface group of the palette.

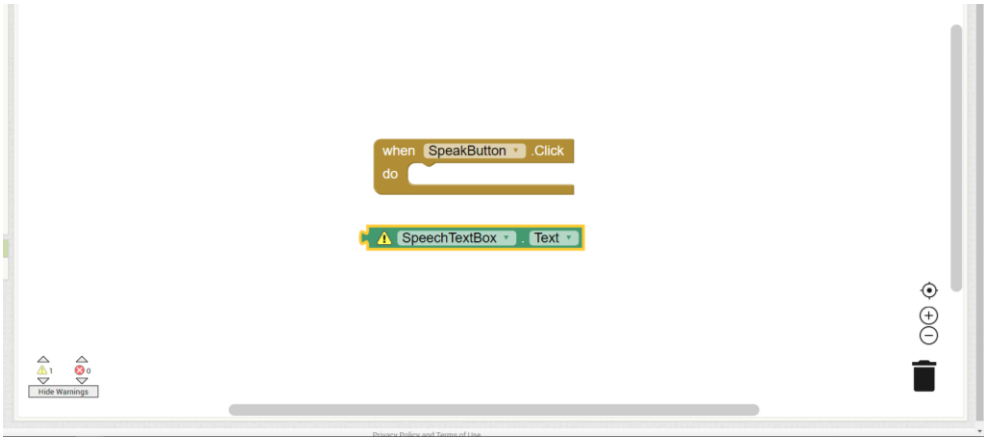
TRY IT OUT: Look through other component groups in the Palette. Can you find a component that can speak text given to it aloud?

There is a component called TextToSpeech under the *Media* group in the Palette, and its help section says it can speak text given to it aloud, so it is definitely what we need. Drag it to the Viewer to add it to our app.



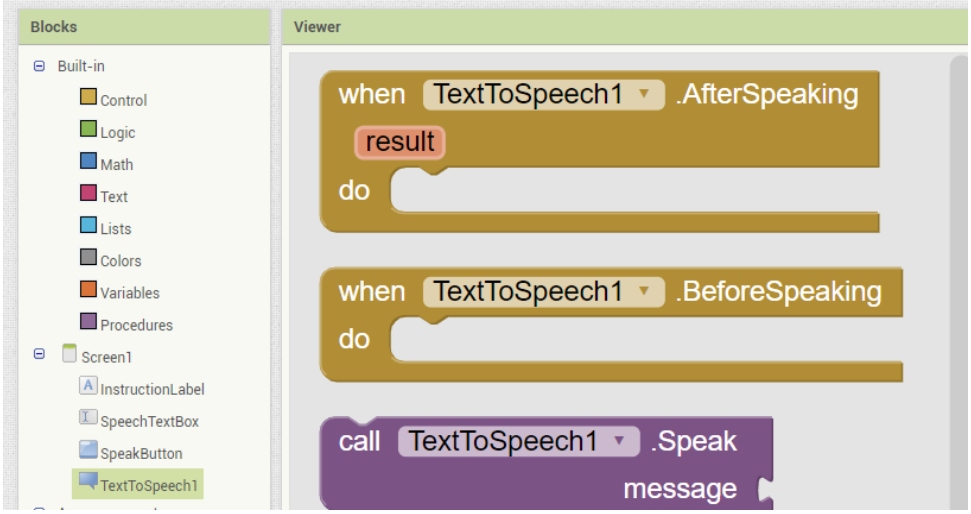
You can see that the TextToSpeech component does not appear on the screen in the Viewer, but instead appears below it, under the **Non-visible components** area. This is because the TextToSpeech component is not a User Interface component; it cannot be interacted with directly with the user's sense organs, and therefore does not need to be shown on the User Interface. Components that appear under the "Non-visible components" area can only be programmed to perform their functions in the Blocks Editor.

We do not need to rename the component as its current name describes exactly what it is used for in the app. Let us switch back to the Blocks Editor to finish programming our app behaviour.

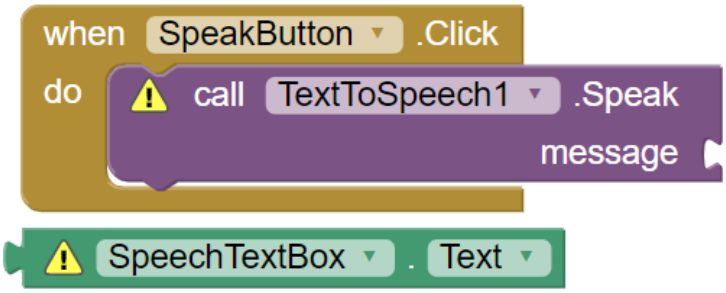


The TextToSpeech component now appears in the Blocks section of the Blocks Editor

For the action **Speak the text**, we need a command block. Let us go through the TextToSpeech component's blocks to see which command block would be used to speak text.

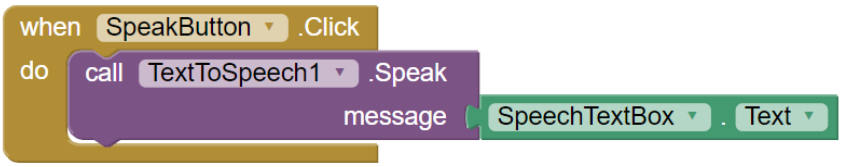


The only command block available for the TextToSpeech component is called "**call TextToSpeech1.Speak**". This is exactly what we need, so drag it to the Viewer and stack it inside the event block added earlier.



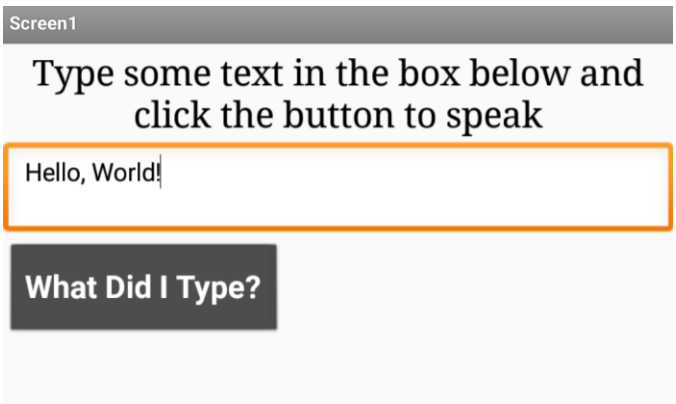
The command block we just added also has a warning displayed on it. This is because it expects an input parameter for its **message** property, but no input has been passed to it. The block's function is to speak text, so we need to give it the text to speak.

We already retrieved the text we want to speak earlier from SpeechTextBox, so, let us pass this text to the TextToSpeech command block as an input parameter for its message property. Drag the property getter we added to the Viewer and snap it to the input section of the command block.



Great! The warnings are now gone and everything looks fine.

TRY IT OUT:
Connect and run the app on your emulator or Android device, enter some text into the text box and click on the "What Did I Type" button. What happens?
NOTE: Make sure the volume on your computer or Android device is turned up so that you would hear the app speak the text you typed.



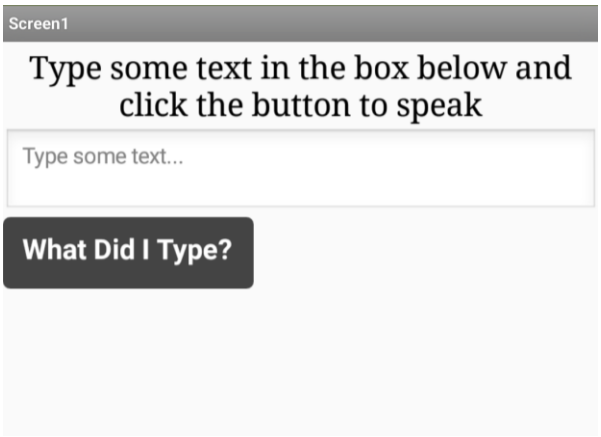
Visual Polish

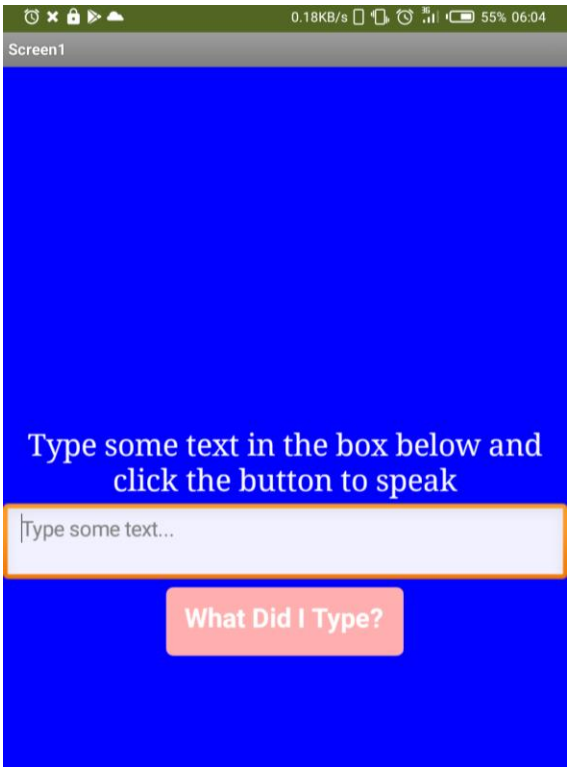
Now that we have learned how program app behave, we need to add an extra step to our app development process. These are the steps in the current version of the process:

- **Determine** all the functions needed in the application.
- **Decide** which components are needed to perform these functions.
- **Design** a User Interface and arrange the components in it.
- **Program** the components to perform their functions when they are interacted with.

When an app is completed with the above steps, it is ready to be used by anyone. However, the users of your app would appreciate it better if your app looks nice and visually appealing.

Consider the following versions of our TextToSpeech app:





Which one would you use more frequently?

In forthcoming modules, we will be using the following updated app development process:

- **Determine** all the functions needed in the application.
- **Decide** which components are needed to perform these functions.
- **Design** a User Interface and arrange the components in it.
- **Program** the components to perform their functions when they are interacted with.
- Visual Polish.

The Visual Polish step would be completed as part of the workbook attached to this textbook. Refer to the workbook for hints on completing the visual polish of the TextToSpeech app.

Summary of Module Three

In this module, you learned about what App Behaviour is and the relationship between App UI and Behaviour. Also, you learned that blocks are used to write code to program app behaviour in App Inventor and that the Blocks Editor is used to program app behaviour in App Inventor.
Finally, you learned about the different types of blocks that can be used to program app behaviour in App Inventor, how to add blocks and scripts to program behaviour in App Inventor and the importance of adding visual polish to completed apps.

MODULE FOUR

TEXT MESSAGING

Module Objectives

At the end of the module, the students should be able to:

- **use the Texting component to carry out operations involving text messaging on Android devices.**
- **package App Inventor projects as standalone applications and install them outside the MIT AI2 Companion app on Android devices.**
- **use blocks from event parameters for behaviour programming in the Blocks Editor.**
- **carry out text-related operations in App Inventor using the 'Text' built-in blocks category.**

Introduction to Text Messaging (SMS)

Long before the advent of the internet on mobile phones, people communicated with one another through text messages (SMS) and phone calls. The internet is surely useful as a means of communication across long distances, but there are situations when access to the internet is not readily available; the connectivity in the area at that time may be poor, or one might not have enough data (megabytes) to access the internet on a given network. The person you are trying to communicate with might not even be online at that moment. What, then, does one do in such a situation? You guessed right; fall back to the traditional means of communicating via text messages and phone calls.

A very useful application of the reliability of communication over SMS in the real world is in SOS apps. SOS (which means "Save Our Soul") is a code for emergency, and SOS apps are used to let loved ones know when we are in a dangerous situation. Every mobile phone has an SOS feature that allows you to dial any number designated as an "emergency number" at any time, even if you do not have a network signal on your cell phone.

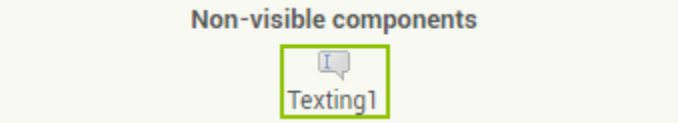
TRY IT OUT: Put your mobile phone on Airplane Mode, and then try to dial any emergency number (112, 911, 999). Did the call go through?

SOS apps are effective and successful because they allow you to send an SMS and make phone calls to desired phone numbers without prior user interaction; that is, the user does not have to open the app and interact with its UI in order to contact loved ones in an emergency.

In this module, we will build different versions of an SOS application. We will start with a simple app that allows the user to click on a button to send a predefined message to a predefined phone number, after which we will modify it to automatically detect when text messages are received by the phone and read them out loud (this is useful for situations like for instance, when someone is driving and cannot take focus off the wheels and road to manually read text messages).

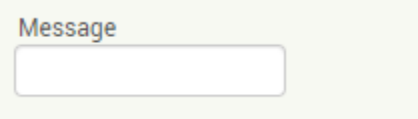
The process of configuring a mobile phone to send and receive text messages to and from phone numbers is complicated, but App Inventor helps us to bypass this complexity with one of its components, the **Texting** component.

The Texting Component



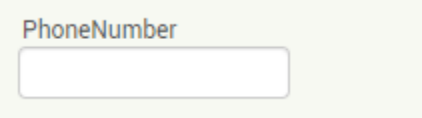
The Texting component is a non-visible component on App Inventor that handles every operation related to SMS/Text Messaging on an Android application. Like every component, it comes with a set of properties and blocks that can be modified and programmed to make it perform its functions.

The 'Message' property



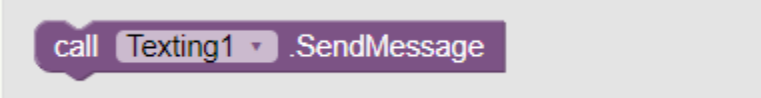
The Texting component has a message property whose value is the text message that would be sent when the component performs its function.

The 'PhoneNumber' property



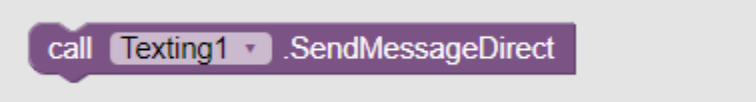
The PhoneNumber property contains the phone number that the text message would be sent to when the component performs its function.

The 'SendMessage' command block



The SendMessage command block is used to instruct the Texting component to perform its main function, and it is to send a text message (which is the value of its Message property) to a phone number (which is the value of its PhoneNumber property).

The 'SendMessageDirect' command block

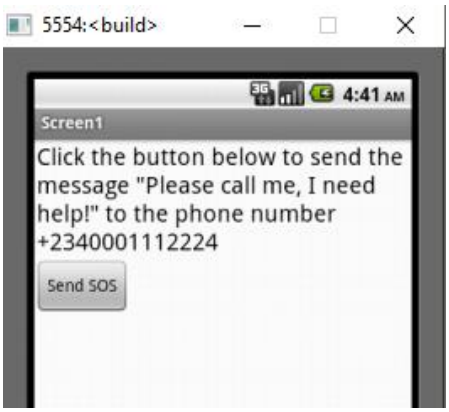


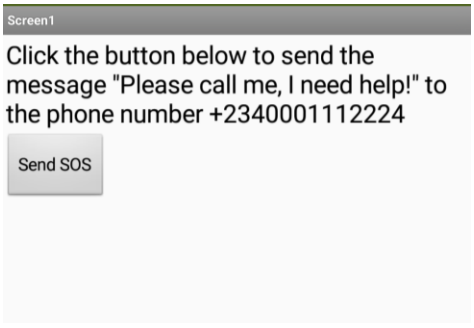
The SendMessageDirect command block is used to instruct the Texting component to send a text message (which is the value of its Message property) to a phone number (which is the value of its PhoneNumber property) directly, without user interaction.

The difference between the SendMessage and SendMessageDirect command blocks is that the SendMessage block opens up the Android device's default messaging app with a message window and populates the message and phone number fields with the values from the corresponding properties from the Texting component, which allows the user to customize the message and phone number, or discard the message if they decide not to send the message again. The SendMessageDirect block, on the other hand, sends the text message directly to the phone number, meaning that once the block is executed, the user can no longer customize the message and phone number.

With the knowledge of these properties and blocks of the Texting component, we can create a basic SOS app that allows the user to click on a button to send a predefined message to a predefined phone number.

Module Project 1: SOS App





NOTE: The phone number above reads "+2340001112224". This is not a real phone number; it is just for demonstration purposes. When building the app, change this to the phone number on the second Android device accessible to you, as you would need to test that your app works by sending real text messages to a real phone number.

The first step is to **determine all the functions needed in the application**. All we need to do in this app is to send a text message to a phone number, so let us note that down.

- Send a text message to a phone number.

The next step is to **decide which components are needed to perform these functions**. We now know that the component that can perform the function of sending text messages is the *Texting* component.

Before the text message can be sent, the user needs to click on a button which, from the images of the final version of the app above, says "Send SOS". The app also displays texts to let the user know which message and phone number would be used by the Texting component when the button is clicked. Therefore, we would also need a Button and a Label component.

- 1 Texting component.
- 1 Button component.
- 1 Label component.

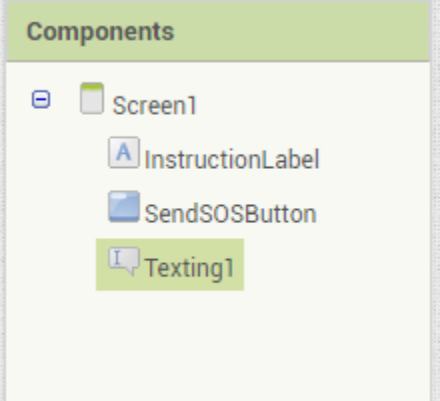
It is now time to design the user interface of the app. Create a new project on App Inventor and name it "SOSApp".

TRY IT OUT: Go ahead and design the user interface for the SOS app, using the images of the final version of the app above as a guide. Here is a summary of the things you need to do:

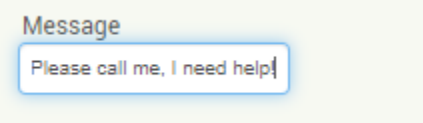
- Add a Label component and change its text and size.
- Add a Button component and change its text.

Do not forget to change the name of the components as you add them, to describe the function of the component in the application.

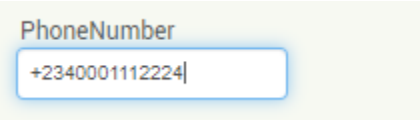
Now that we have our user interface ready, it is time to **program the components to perform their functions as they are interacted with**. Add the Texting component to the app from the 'Social' group in the Palette.



Next, change the value of its Message property to “Please call me, I need help!”.



Change the value of its PhoneNumber property to a phone number in the second Android device that you have access to, for live testing. This phone number should correspond with the one in the InstructionLabel.



Now switch to the Blocks Editor to program the Texting component.
We need just one behaviour in this app:

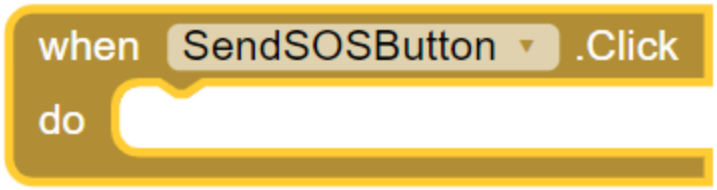
send the Message property of Texting1 to its PhoneNumber property when SendSOSButton is clicked.

We now have to determine if this behaviour is valid by splitting it into events and actions.

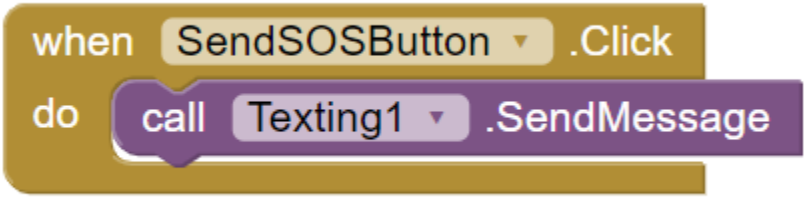
- **When SendSOS Button is clicked** - this is a **click event** that is triggered on the SendSOSButton component.
- **Send the Message property of Texting1 to its PhoneNumber property** - this is an **action** that is performed on the Texting1 component in response to the click event.

There is an event and an action, so the behaviour is valid. Let us proceed!

Add an event handler for the click event by dragging the appropriate block for SendSOSButton to the Viewer.



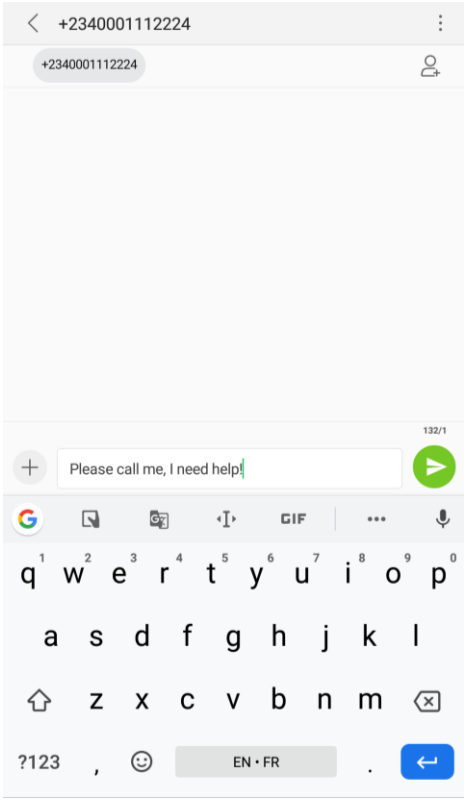
With the event handled, let us program the Texting1 component to carry out the action of sending the text message.



Note that the SendMessage command block does not have any input parameters. This is because the Texting component uses the values of its Message and PhoneNumber properties at the time the block is executed. Therefore, if you need to modify any of the properties before sending the text message, you would have to place property setter blocks before the command block.

Now run the app on an Android device and test it.

TRY IT OUT: Click on SendSOSButton in the app. What happens?

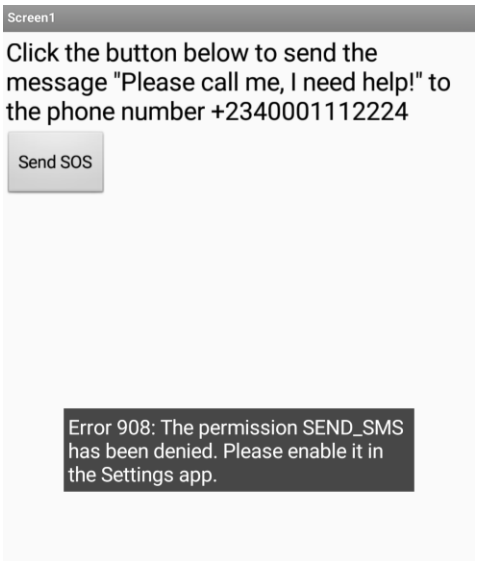


The app opens the default messaging app on the device and with a message box open and prefilled with the text “Please call me, I need help!”.

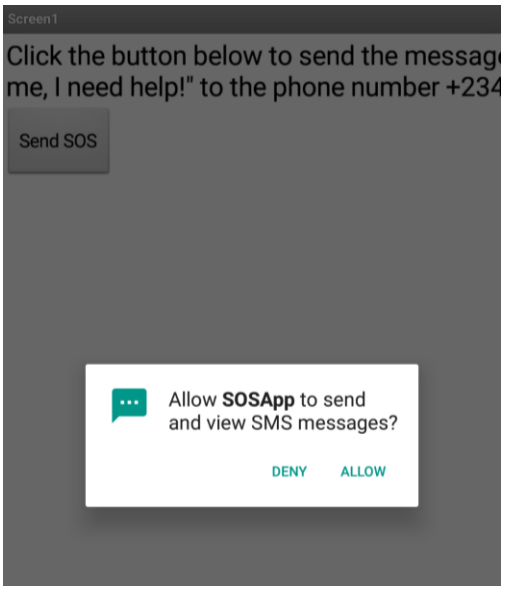
This is, however, not what we need for an SOS app. The point of such an app is to allow the user to send a distress message with the least interaction possible. We, therefore, need to reprogram the app to send the message directly to the phone number.

TRY IT OUT: Swap the SendMessage block with the SendMessageDirectly block and test the app. Does it send the text message to the second Android device without user interaction?

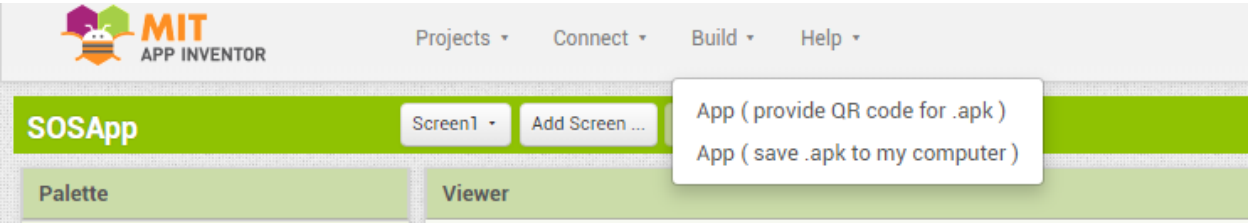
You might get the following error when you click on SendSOSButton.



The action of sending an SMS directly without user interaction is a sensitive one, so the Android system needs the user to give any app that wants to do that the permission the first time such action occurs. Normally, the app should ask you for that permission the first time you click on SendSOSButton:



However, you did not see this because you are running your app in the MIT AI2 Companion app.
To fix this issue, we have to build the app as a separate, standalone application and install it directly on the Android device. Click on *Build* at the top of the App Inventor project screen.



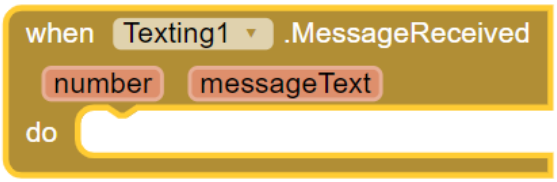
Both options would generate an application installation file with the extension ".apk" (the file would be named [app_name].apk, for example, SOSApp.apk). The first option would provide a QR code (similar to the one that appears when you connect to the AI Companion app) which can be scanned to download the installation file, while the second option would save the installation file to your computer; you can then transfer the file to your Android device and install it like any other application.

After installing the app as a standalone application, open it and click on SendSOSButton. Does it ask you for permission to send and view SMS messages?

After granting the app this permission, further clicks on the SendSOSButton would work as expected - send the SOS message directly without further user interaction.

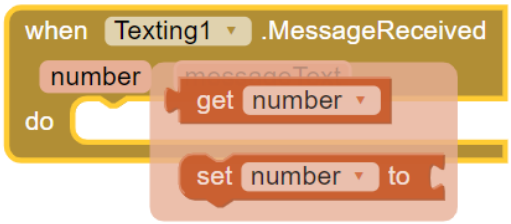
The 'MessageReceived' event handler

It was mentioned earlier that the Texting component is also capable of detecting when text messages are received on a device. Just as in the alarm clock example that was discussed in the last module, an Android device broadcasts an event when it receives an SMS from another device. The Texting component can be programmed to listen to this event with its 'MessageReceived' event block.



This block looks a bit different from the event blocks that we have seen so far, because it has two properties. These properties are called **event parameters**. Event parameters contain values for extra information about an event that could be useful in the programming of the reaction of the app to that event. For the MessageReceived event of the Texting component, the **number** event parameter contains the phone number that sent a text message to the device, and the **messageText** event parameter contains the text message that was received.

An event parameter has getter and setter blocks that can be used to retrieve and modify its value. Hover your mouse over a parameter to see its blocks, and drag to use them in your scripts.



Let us add the ability to detect received messages and read them out to our SOSApp.

Module Project 2: MessageReader App

We want to modify our SOS App to detect when text messages are sent to the device on which it is running, and read the messages (including the sender's phone number) out. The app will still have the initial feature of sending an SOS message.

The app will read out the text message in this format:

"You have received the following text message from [phone number]: [text message]"

For example, if a text message that reads "I'm on my way" is received from +2348031112223, the app should read the following out aloud:

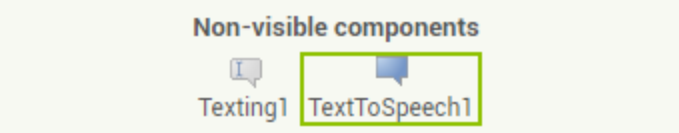
"You have received the following text message from +2348031112223: I'm on my way"

Even though we are adding a new feature to an existing app and not building a new one from scratch, we will still follow the same app development process.

The first step is to **determine all the functions needed in the application**.

- Read out received text messages.

The next step is to **decide which components are needed to perform these functions**. The only additional component needed to perform the above function of reading out text is the *TextToSpeech* component which we learned in the last module. Add a TextToSpeech component to your app in the Designer Window.



Since we did not add any additional UI component, let us go ahead to **program the components to perform their functions** in the Blocks Editor.

We need one additional behaviour in the app:

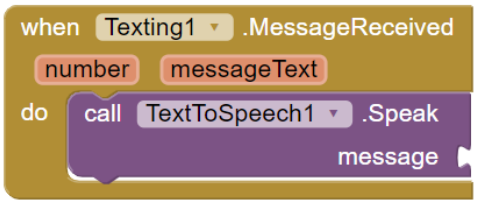
read out the message and phone number when a text message is received on the device

As usual, we will split this behaviour into events and actions to determine its validity:

- **When a text message is received:** this is an **event** that can be handled by the 'MessageReceived' event block of the Texting component.
- **Read out the message and phone number:** this is clearly a text to speech **action** that would be performed by the TextToSpeech component.

The behaviour is valid, so we can go ahead to program it in the Blocks Editor.

Add the 'MessageReceived' event block of the Texting component, and the 'Speak' command block of the TextToSpeech component.



The 'Speak' block requires an input parameter, which is the text that is to be converted to speech by the TextToSpeech component.

Remember the format in which we want to read out the text message:

"You have received the following text message from [phone number]: [text message]"

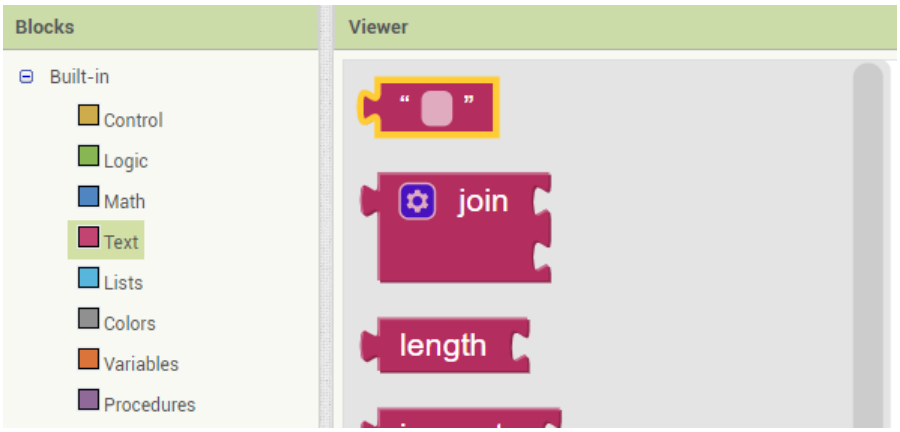
This format can be broken down into four parts:

- "You have received the following text message from "
- [phone number]

- ": "
- [text message]

[Phone number] and [text message] can be gotten from the event parameters of the 'MessageReceived' event handler, while we have to manually type the first and third parts of the text.

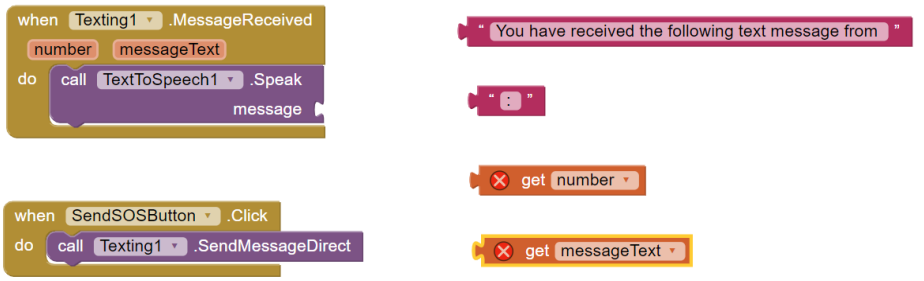
App Inventor has a built-in block category that contains blocks for dealing with text. The very first block in the category can be used to type text that can be passed as input to other blocks.



Drag the block to the Viewer and type in the first part of the text above.

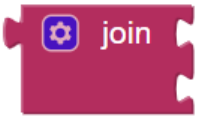
Do the same for the third part of the text.

For the second and fourth parts of the text, drag the getter blocks from the corresponding event parameters of the 'MessageReceived' event handler.

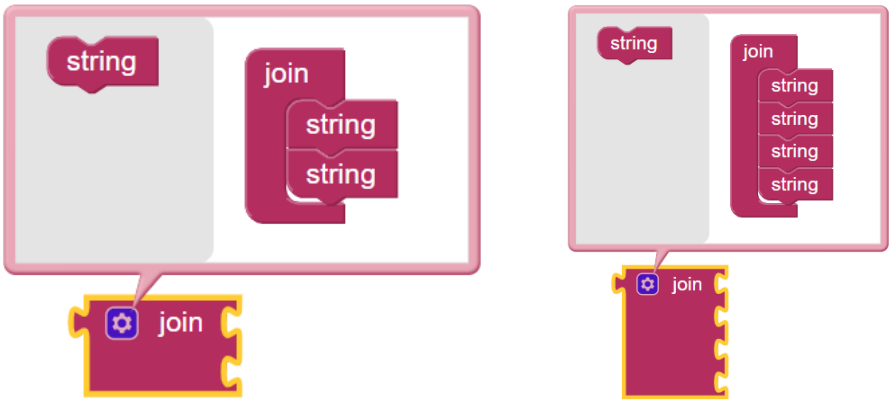


We now have the four parts of the text that the app should read out when a text message is received, but the TextToSpeech's 'Speak' command block accepts only one input parameter. We need to find a way to combine these four blocks into one text and pass the single block to the TextToSpeech's 'Speak' block.

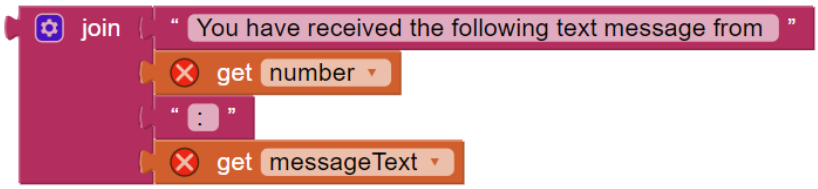
The built-in Text block category has a 'join' block that can be used to join different text blocks into one.



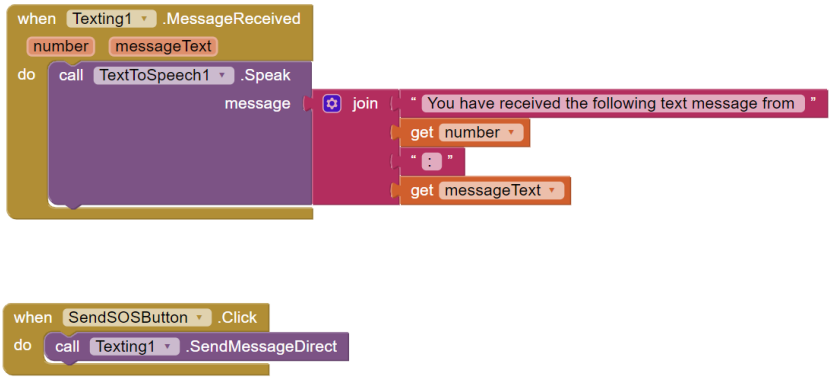
The 'join' block has two input parameters by default, but you can add as many parameters as you want, depending on the number of text blocks you want to join together. In this case, we want to join four text blocks together, so click on the settings icon on the 'join' block and drag the 'string' block from the left side into the join block on the right side twice.



Then snap the four text blocks into the join block in the order in which we want them joined.



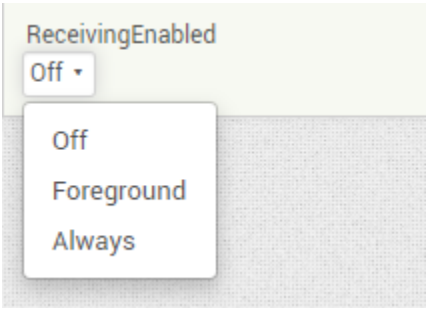
Finally, pass the join block as an input parameter to the TextToSpeech's 'Speak' block.



Now test the app by sending a text message from the second Android device to the one on which you are running the app through the MIT AI2 Companion. Does it read the text aloud?

The app is expected to read the text message received and the phone number in the format we programmed it to above, but nothing happens when a text message is received on the device. Let us fix this.

Switch back to the Designer Window, select the Texting component and look at its properties; there is a *ReceivingEnabled* property whose value is "off". This is the property that determines whether your app will be able to detect incoming text messages or not.



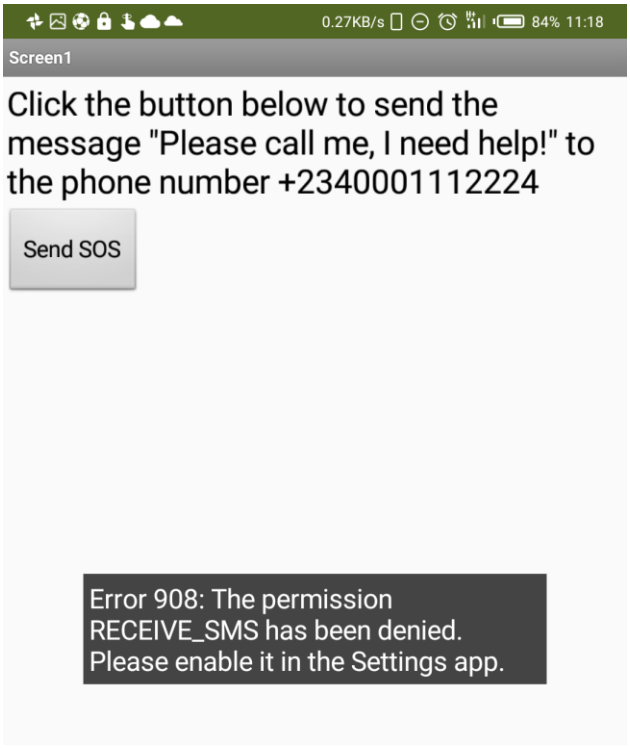
The other two possible values for the property are *foreground* and *always*. When the property is set to *foreground*, the app will only detect incoming messages when it is running. When the property is set to

always, the app will detect incoming messages both when it is running and when it is not; if a text message is received when it is running, the component will execute the script in its 'MessageReceived' event handler, and if a message is received when it is not running, a notification would be shown on the phone. If the user clicks on the notification, the app is opened and the component executes its 'MessageReceived' event handler's script.

NOTE: The behaviour of the 'MessageReceived' event handler when the *ReceivingEnabled* property is set to *always* is not consistent on every Android device, so we will only use the *foreground* property value in this textbook.

TRY IT OUT: Set the property of the *ReceivingEnabled* property of the Texting component to *foreground* and test your app. Does it now read the text aloud?

If you are running the app through the MIT AI2 Companion app, you would get the following error:



This is because the act of detecting incoming messages is considered sensitive. Follow the steps we took to fix a similar error in the previous section and grant the app permission if required. The app would now be able to read the text aloud when a text message is received on the device.

Summary of Module Four

You have learnt that text (SMS) messaging is useful for situations when internet connectivity is unavailable and communication has to be made. App Inventor provides a *Texting* component which can be used to

send text messages both directly and indirectly through its *SendMessageDirect* and *SendMessage* blocks. The Texting component can also be used to detect incoming text messages with its *MessageReceived* event handler. The *PhoneNumber* property of the Texting component contains the phone number to which a text message is to be sent, its *Message* property contains the text message to be sent, and its *ReceivingEnabled* property determines whether the component is able to detect incoming messages.

You also learnt that actions like sending an SMS without user interaction and detecting incoming messages are considered sensitive, and requires the user to manually grant an app the permission to do so. Apps that require the user to grant a permission should be installed as standalone applications, rather than through the MIT AI2 Companion app.

Then, you learnt that an event block can have *event parameters*, which contain additional information about the event. Event parameters have getter and setter blocks that can be used to retrieve and modify their values.

Finally, remember that the *Text* built-in blocks category in App Inventor provides blocks for handling text operations while the *join* block can be used to join any number of text blocks together into one.

MODULE 5

Introduction to Data Sharing & Images on App Inventor

Table of Contents

Learning Objectives

By the end of this module, the student would be able to:

- Program Android apps to share data with other apps.
- Display and perform operations related to images in App Inventor.
- Progressively add features to Android apps to enhance user experience.

Module Prerequisites

- A physical Android device is needed to test the functionality of the apps that will be built in this module.

- A reliable internet connection should be made available on the student's laptop/computer, and if using an Android device for app testing instead of the emulator, both the computer and the Android device should be connected to the same Wi-Fi network.
- A Google (Gmail) account is needed by the student to create projects and make Android apps on the App Inventor platform. Login details should either be provided to the student, or the login should be done for the student when needed.

Data Sharing in Android Apps

So far, we have built Android apps that have useful applications in the real world. The TextToSpeech app built in module 3 can be used to listen to large amounts of text instead of reading them, and the MessageReader app built in module 4 can be used by people who want to practise road safety and abide by the 'no texting while driving' rule.

A common pattern that can be observed in these apps, is that they are able to carry out their useful functions because their components share data among themselves.

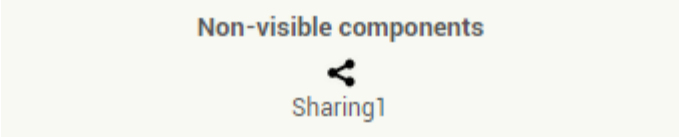
In the TextToSpeech app, the SpeechTextBox component shares its text property through its 'Text' property getter block with the 'Speak' action block of the TextToSpeech component. Without this sharing of data between these components, the TextToSpeech app would not work the way it does.

Likewise, in the MessageReader app, the Texting component shares the phone number and message of text messages it receives (through event parameters in its 'MessageReceived' event handler) with the 'Speak' action block of the TextToSpeech component, thereby making it possible for the app to read out text messages received on the Android device.

This act of sharing data between components is essential to the functioning of apps, but sharing of data is not limited to components within an app alone; an app might need to share data with other apps on the same Android device, or with other devices through wired or wireless connections. For example, we might want to share the text message received in the MessageReader app with a contact on a social app (Twitter, WhatsApp, etc) on our phone.

We got a glimpse of sharing data between multiple apps in the MessageReader app, when the use of the 'SendMessage' block of the Texting component resulted in the app opening another app, the default text messaging app on the device, and sharing the text message and phone number with it. In this module, we will learn more about sharing different types of data, which includes messages and files (image, audio, video, document, etc), from our app to other apps on an Android device.

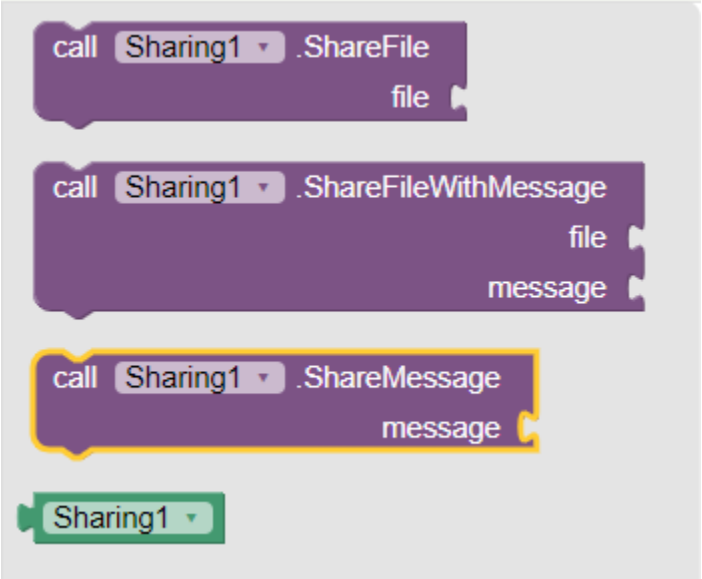
The Sharing Component



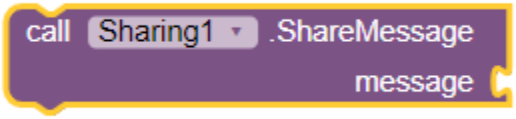
The Sharing component is a non-visible component on App Inventor that handles every operation related to message and file sharing.

Unlike all the components that we have used so far, the Sharing component has no properties.

All but one of its blocks are command blocks; the only property getter block that it has is one that reports the component itself.



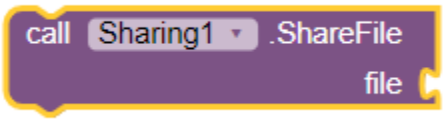
The ‘ShareMessage’ command block



The ShareMessage command block is used to instruct the Sharing component to share a message, usually in the form of text, with other applications on an Android device. It accepts one input parameter, which is the message to be shared.

When this block is executed, the device displays a list of all installed applications that can handle text data, allowing the user to choose one to share the message to. If there is just one app installed on the device that can handle text data, the app is opened directly and the message passed to it.

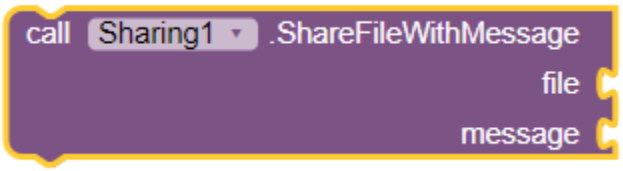
The 'ShareFile' command block



The ShareFile command block is used to instruct the Sharing component to share a file with other applications on the Android device. There are different types of files that can be handled on an Android device, including images, audio, video, and documents. The ShareFile block accepts one input parameter, which is the file to be shared.

Similar to the ShareMessage block, when this block is executed, the device displays a list of all installed applications that can handle the type of file being shared for the user to choose from. If there is just one app that can handle the file being shared, the app is opened directly instead.

The 'ShareFileWithMessage' block

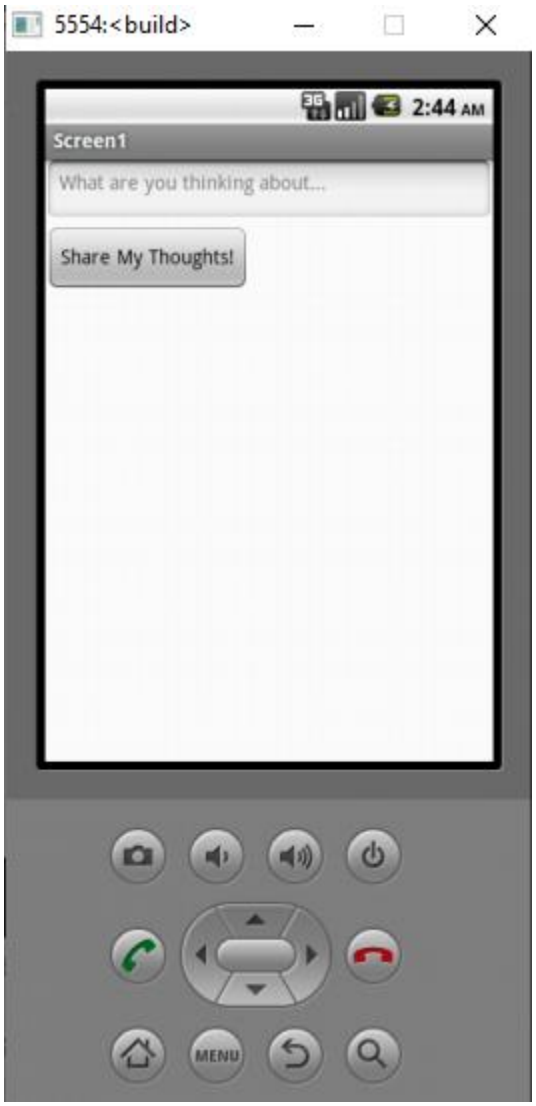


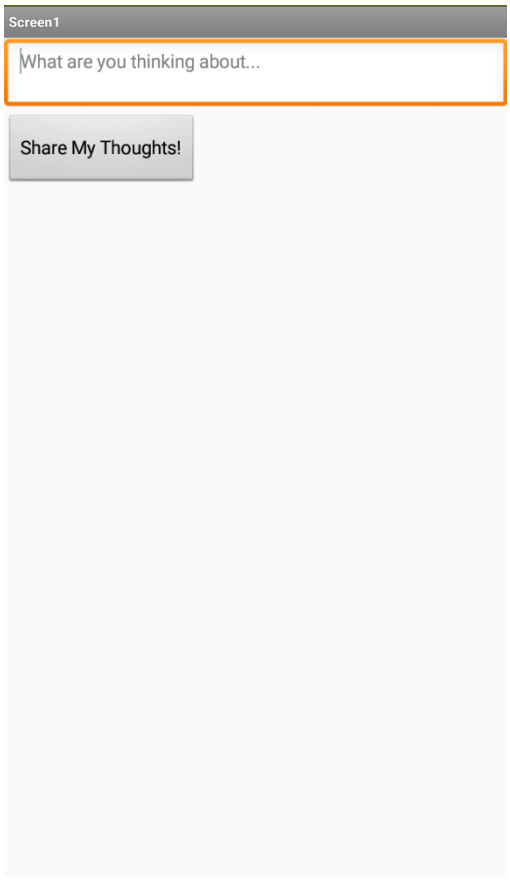
The ShareFileWithMessage block is a combination of the first two blocks. It is used to instruct the Sharing component to share a file and a message together with other applications on the Android device.

When this block is executed, the device displays a list of all installed applications that can handle **both** a message and the type of file being shared. If there is just one app that can handle both a message and the file being shared on the device, it is opened directly when the block is executed.

Let us build an app that can be used to share thoughts, or small notes, with other applications on the same device.

Module Project 1: MyThoughts App





The MyThoughts app is a simple app that allows the user to enter their thoughts and then share them to other apps on the same Android device.

The first step in building this app is to **determine all the functions needed in the application**. We identified the two functions needed in MyThoughts in the above paragraph:

- Allow the user to enter their thoughts (text).
- Allow the user to share these thoughts with other applications on the device.

Next, we have to **decide which components are needed to perform these functions**.

For the first function noted above, we need a component that allows the user to enter text.

- 1 TextBox.

For the second function, we need the Sharing component to share the user's thoughts with other applications. A Button component is also needed for the user to initiate the sharing action.

- 1 Sharing component.

- 1 Button.

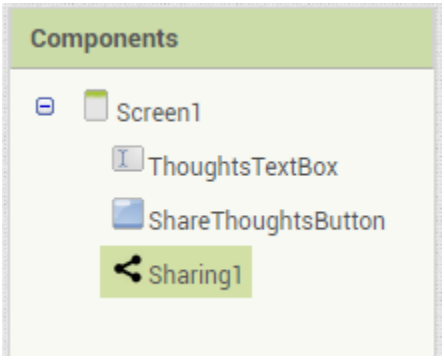
It is now time to design the user interface of the app. Create a new project on App Inventor and name it "MyThoughts".

TRY IT OUT: Go ahead and design the user interface for the MyThoughts app, using the images of the final version of the app above as a guide. Here is a summary of the things you need to do:

- Add a TextBox component and change its width and hint.
- Add a Button component and change its text.

Do not forget to change the name of the components as you add them, to describe the function of the component in the application.

Now that we have our user interface ready, it is time to **program the components to perform their functions as they are interacted with**. Add the Sharing component to the app from the 'Social' group in the Palette.



Now switch to the Blocks Editor to add behaviour to the MyThoughts app.

We need just one behaviour in this app:

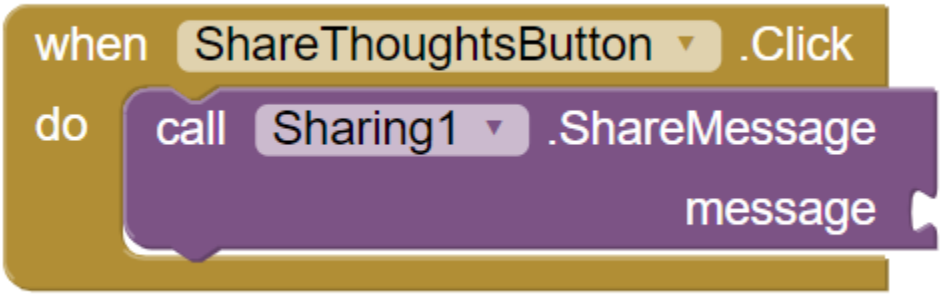
share the text in ThoughtsTextBox to other apps in the device when ShareThoughtsButton is clicked.

As usual, let us split this behaviour into events and actions to determine its validity:

- **When ShareThoughtsButton is clicked:** this is a **click event** that is triggered on the ShareThoughtsButton component.
- **Share the text in ThoughtsTextBox to other apps…:** this is an **action** that is performed by the Sharing component using data from the ThoughtsTextBox component.

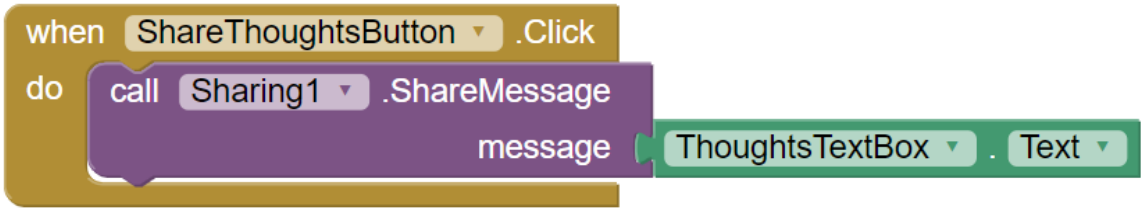
We can go ahead to program this behaviour into our MyThoughts app.

Add the 'Click' event handler of the ShareThoughtsButton and the 'ShareMessage' command block of the Sharing component.

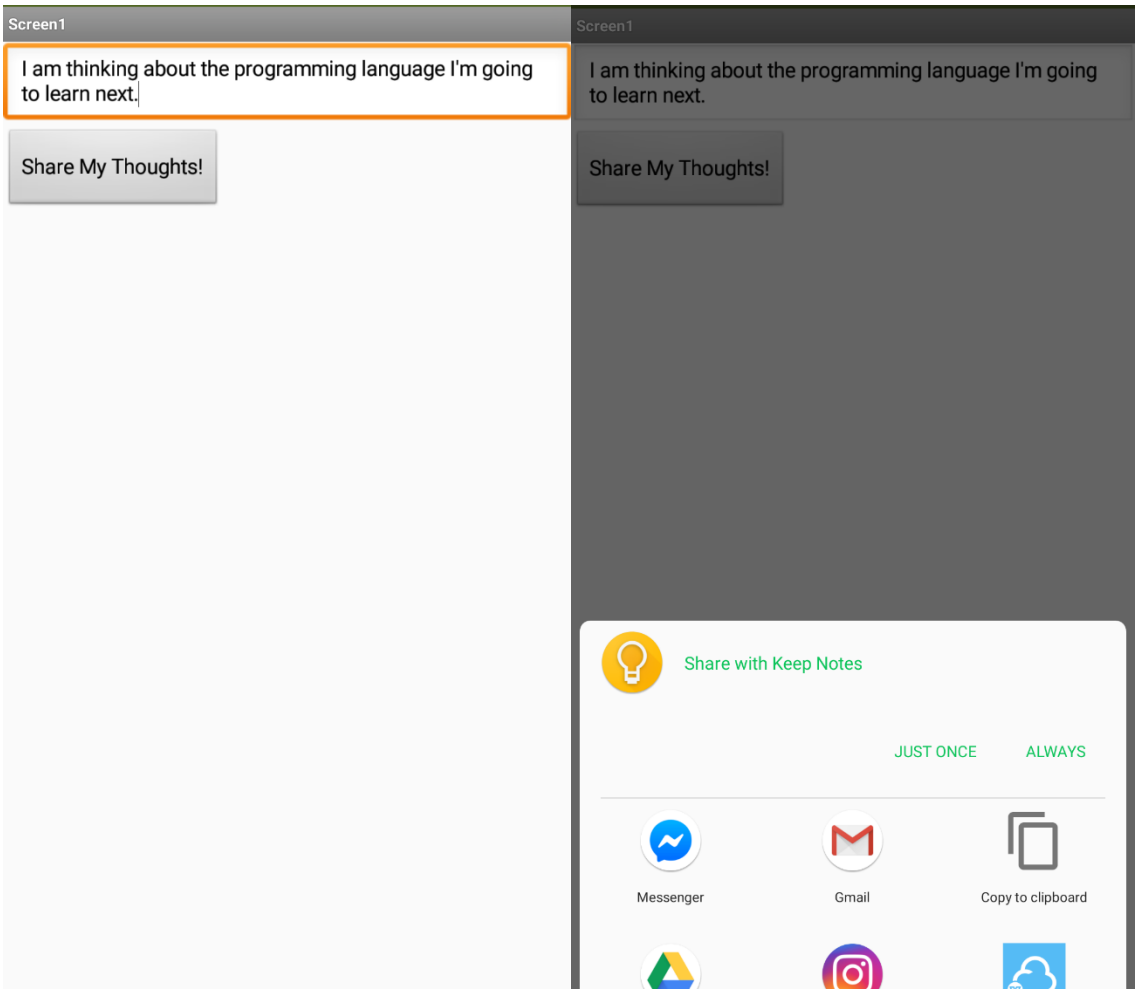


The 'ShareMessage' block needs an input parameter for the message that is to be shared with other applications. In this app, the message that is to be shared is the text typed by the user into the ThoughtsTextBox. Therefore, we need to get the text from ThoughtsTextBox using the appropriate property getter block.

Snap this block into the 'ShareMessage' block.

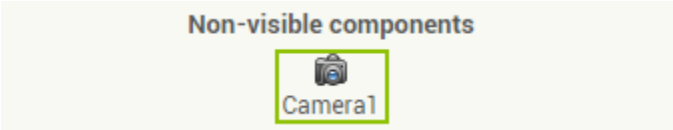


Test the app on an Android device by typing some text into the textbox and clicking on the share button.



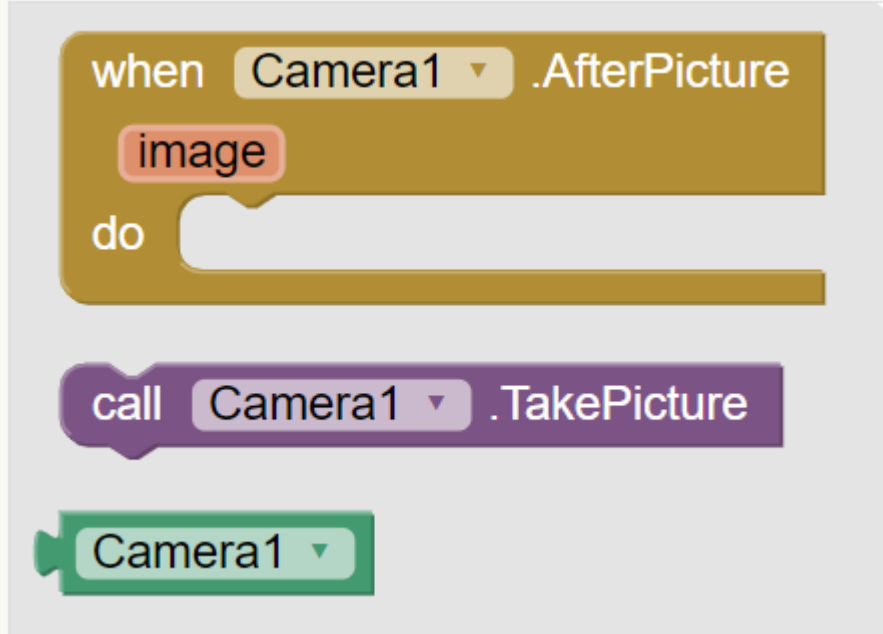
The Sharing component allows us to share more than just text messages; it can be used to share different types of files. We will learn about these different file types in subsequent modules; for now, we will start with images and modify our MyThoughts app so that the user will be able to take an image with the device camera and share alongside the text message. For this, we would need the *Camera* component.

The Camera Component

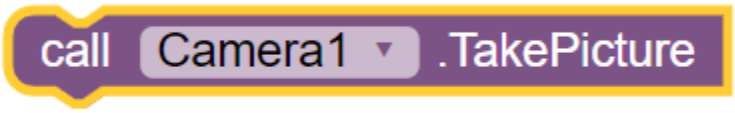


The Camera component is a non-visible component that uses a device's camera to capture images. Like the Sharing component, it has no properties.

It has one event block, one command block and one property getter block which reports the component itself.

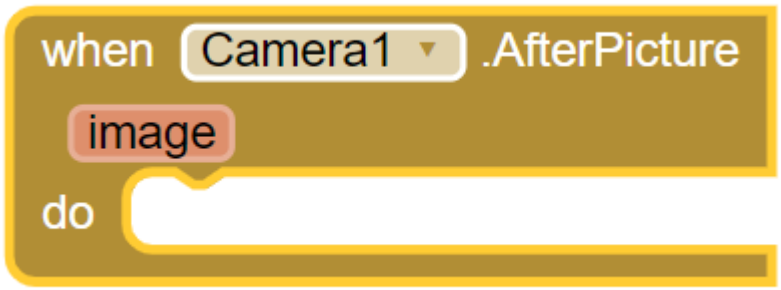


The ‘TakePicture’ command block



When this block is executed, the camera application on the device is opened and the user can take a picture. The picture taken through this command block is then returned for use in the app through an event handler in the Camera component.

The ‘AfterPicture’ event handler



The existence of the Camera component in an application indicates that the application needs to use the images captured through the component. When a picture is taken using the Camera component, the 'AfterPicture' event is triggered on the component and the image captured using the component is passed as a parameter to the event handler.

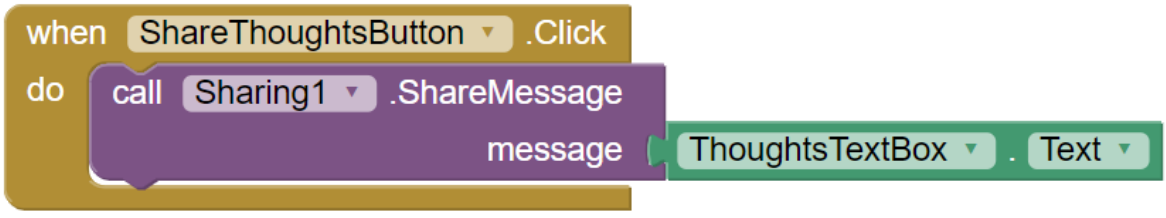
Let us modify the MyThoughts app to allow the user to take a picture when SendThoughtsButton is clicked, and then share the image together with the message typed into ThoughtsTextBox to other applications on the device. Add the Camera component to your app from the 'Media' group in the Palette.

We now have two behaviours in our app: a slight modification of the first one, and a new one.

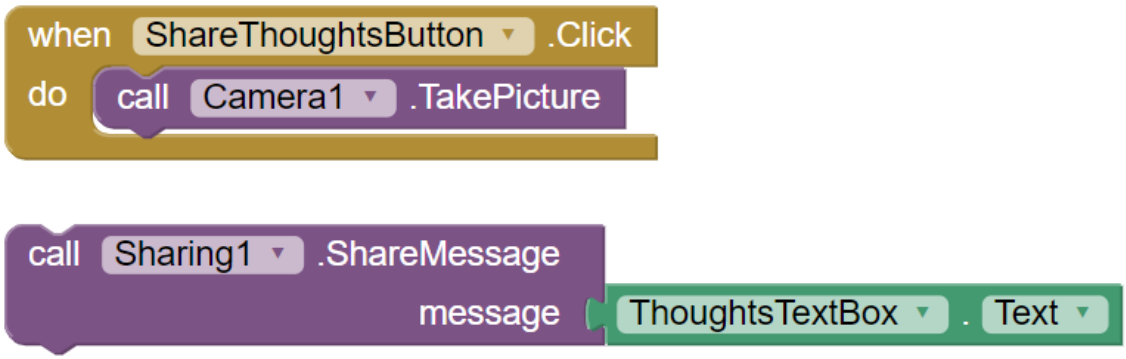
- **Behaviour 1:** launch the camera when ShareThoughtsButton is clicked.
- **Behaviour 2:** when an image is captured, share the image from the camera together with the text in ThoughtsTextBox to other apps.

We will program each behaviour separately, starting with Behaviour 1.

This is the current code for the first behaviour:

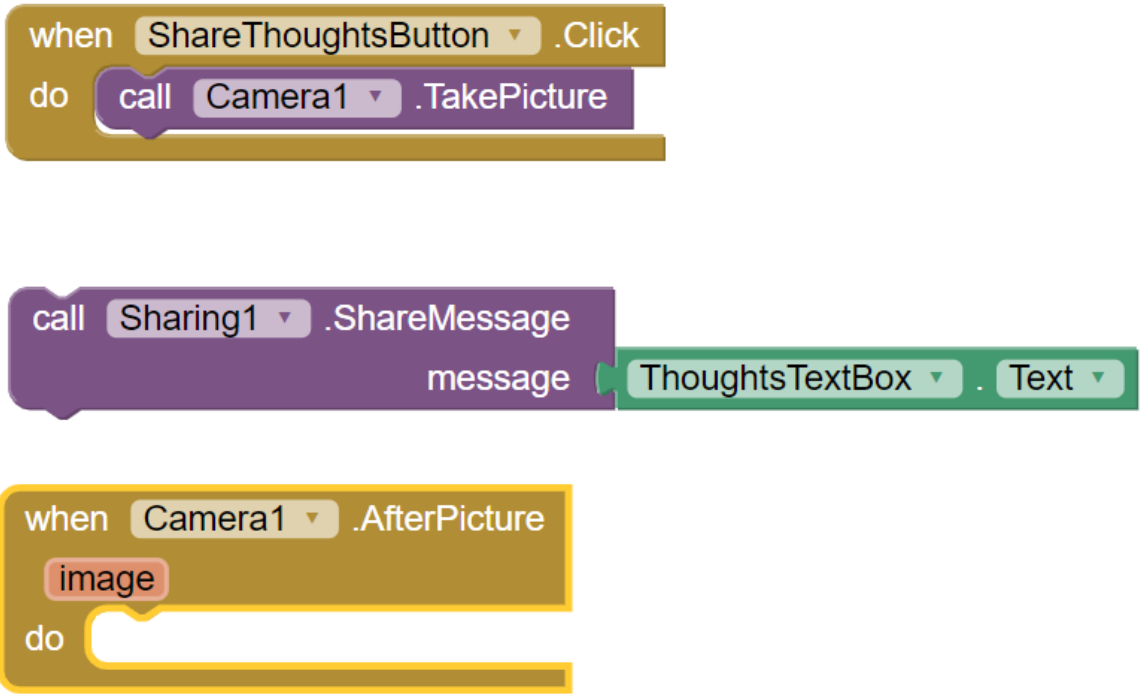


Detach the 'ShareMessage' block from the click event handler, and add the 'TakePicture' command block from the Camera component to the event handler.



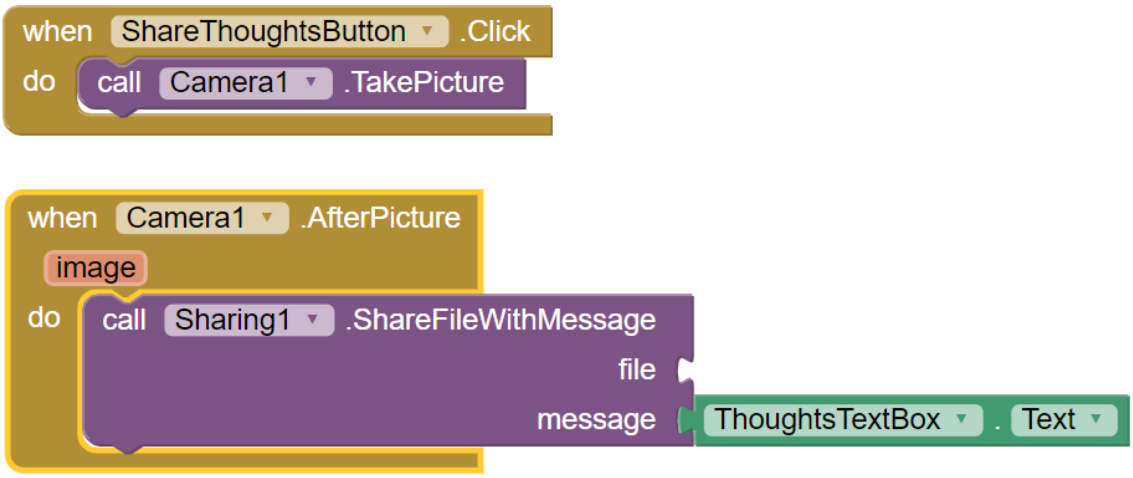
TRY IT OUT: Now test this behaviour by clicking on ShareThoughtsButton on your device. Does it open the camera app for you to take pictures? What happens when you take a picture and return to the app?

For the second behaviour, add the 'AfterPicture' event handler to the Viewer from the Camera component's list of blocks.

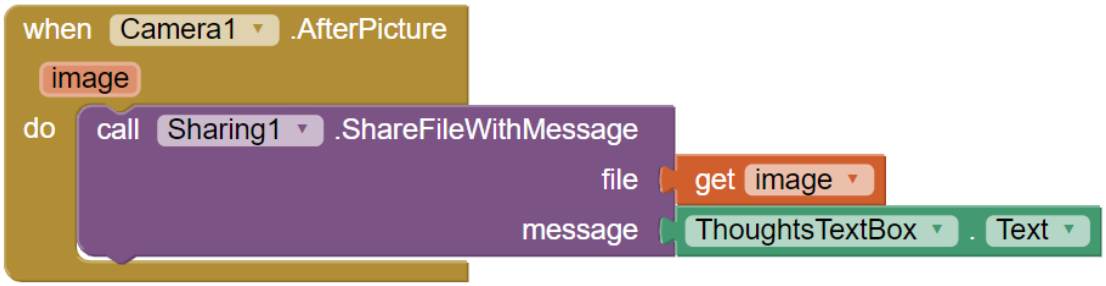


We can now attach the 'ShareMessage' block to the 'AfterPicture' event handler to perform the sharing operation. However, we want to share both the text from ThoughtsTextBox and the image from the

Camera component, so replace the ShareMessage block with the 'ShareFileWithMessage' block from the Sharing component.



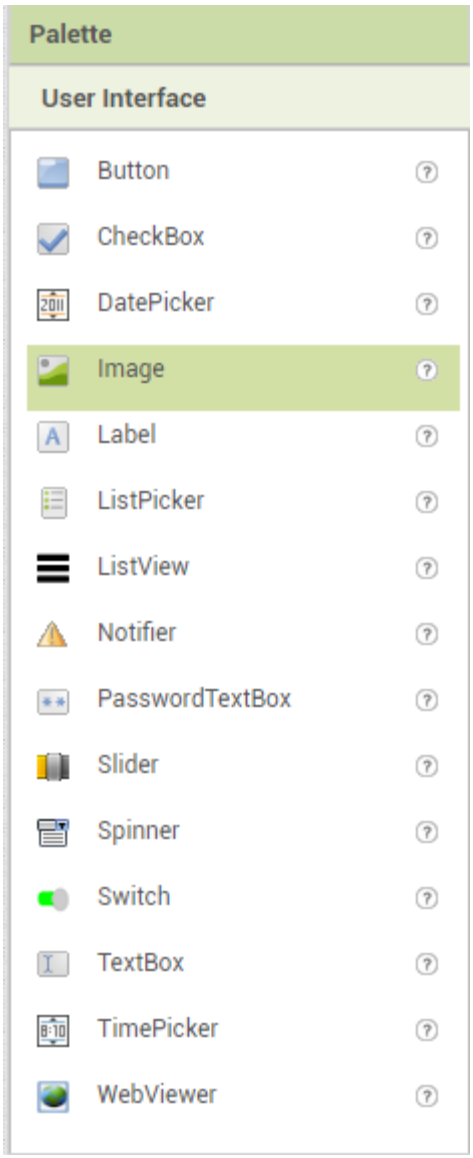
Finally, get the image from the event parameter and pass it as an input to the ShareFileWithMessage *file* input parameter.



If you test the app now, clicking on ShareThoughtsButton launches the camera, and taking a picture shares the image together with your thoughts directly. This is nice, but it would be nicer if you could allow the user to see the picture they took in the app before choosing to share it to other apps. This gives the user a chance to take another picture if they do not like the existing one, and enhances the overall user experience. Let us modify MyThoughts to include this.

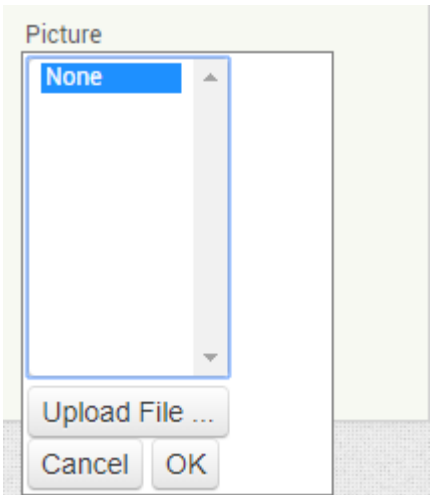
App Inventor has an *Image* component that can be used to display images in an app's user interface.

The Image Component



The Image component is a visual component whose primary function in App Inventor is to display images. It is located in the User Interface group under the Palette.

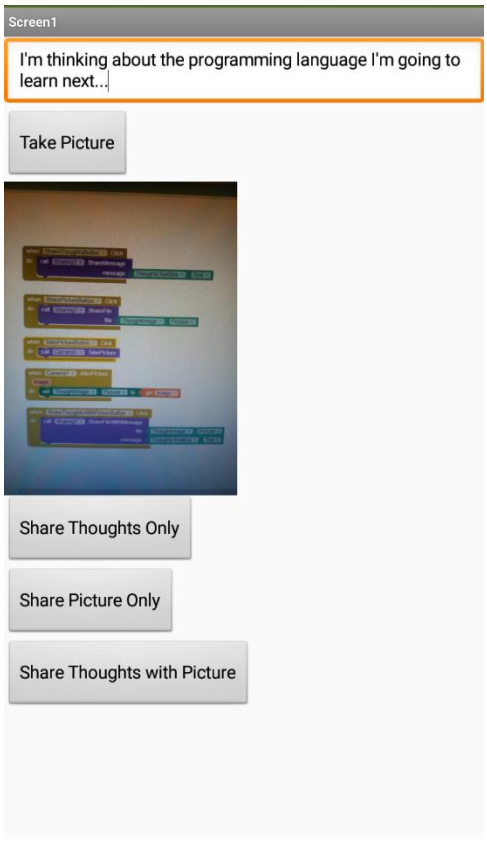
The ‘Picture’ property



The Picture property is the most important property of the Image component. It is used to specify which image should be displayed in the component. In the Designer Window, images can be uploaded from the computer storage, via the Media section, to display on the screen in the Viewer. This property can be modified using the 'Picture' property setter block in the Blocks Editor. The Image component has no event and command blocks.

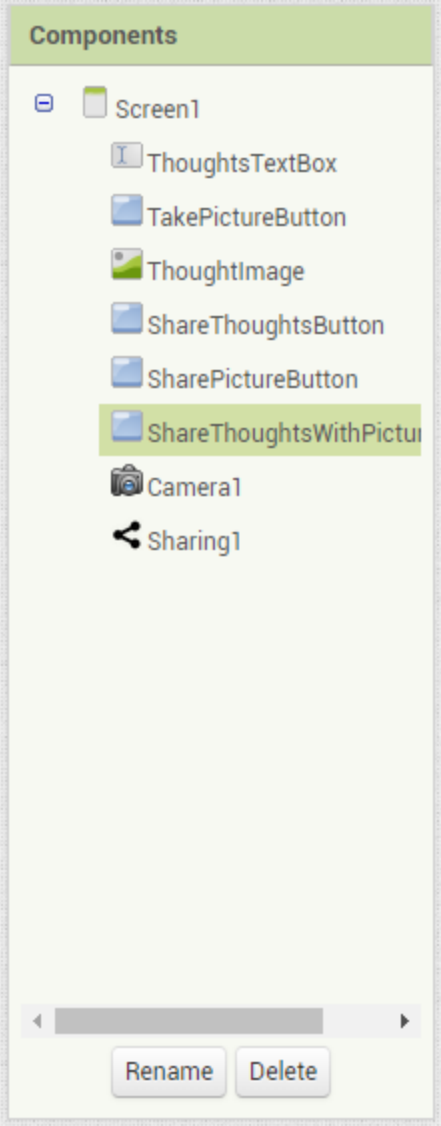
Let us modify MyThoughts to include a button for the user to capture an image from the camera and display it in an Image component on the User Interface. We will then separate the ShareThoughtsButton into three buttons, the first to allow the user to share just the thoughts (text) typed into ThoughtsTextBox, the second to allow the user to share just the image, and the third to allow the user to share both together.

This is what MyThoughts would look like by the time we are done implementing these features:



TRY IT OUT: Go ahead and modify your app's UI to match the image above (Note: the image in the UI is an Image component, and an image appears only because the user has taken a picture with the app on an Android device. An image would not appear when you add the Image component in your own app on App Inventor).

Here is a list of the components that you should have in your app at this stage:

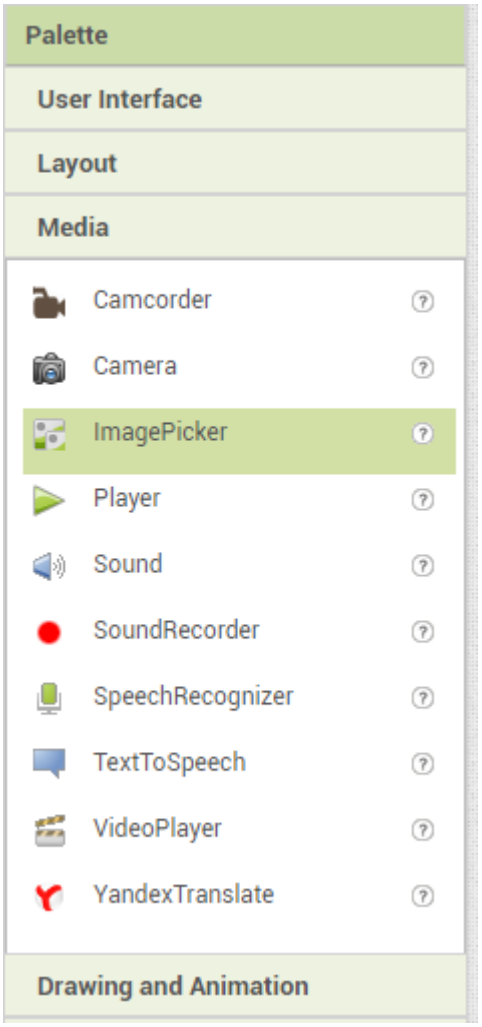


Now switch to the Blocks Editor and program the following behaviours into your app:

- When TakePictureButton is clicked, launch the camera.
- When the 'AfterPicture' event is triggered on the Camera component, set the 'Picture' property of ThoughtImage (the Image component) to the *image* event parameter of the event handler. This should make the Image component display the image captured by the camera in the app, like the app screenshot above.
- When ShareThoughtsButton is clicked, the app should share just the text in ThoughtsTextBox.
- When SharePictureButton is clicked, the app should share just the picture in the Image component. (Hint: use the property getter block of the Image component to get its picture.)
- When ShareThoughtsWithPictureButton is clicked, the app should share both the text in ThoughtsTextBox and the picture in the Image component.

The MyThoughts app now offers a better user experience, but we can improve further. The app currently allows the user to share only pictures taken from the camera; what if the user wants to share images that are already stored on the device gallery or storage? This would mean that the user would have to find another app that affords them that choice, and we do not want users leaving our app for other apps. Let us fix that.

The ImagePicker Component

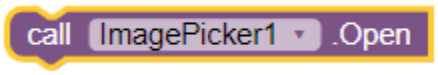


The ImagePicker component is a visible component that displays a button on the User Interface. When this button is clicked, the device's image gallery is opened and the user can select an image.

The 'Selection' property

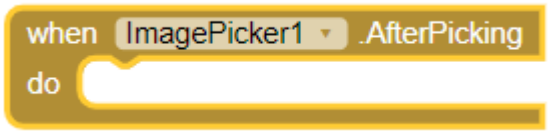
The image selected by the user is stored in the ImagePicker's 'Selection' property. This property is not available in the component's property section in the Designer Window, and is only accessible through a property getter block in the Blocks Editor.

The 'Open' command block



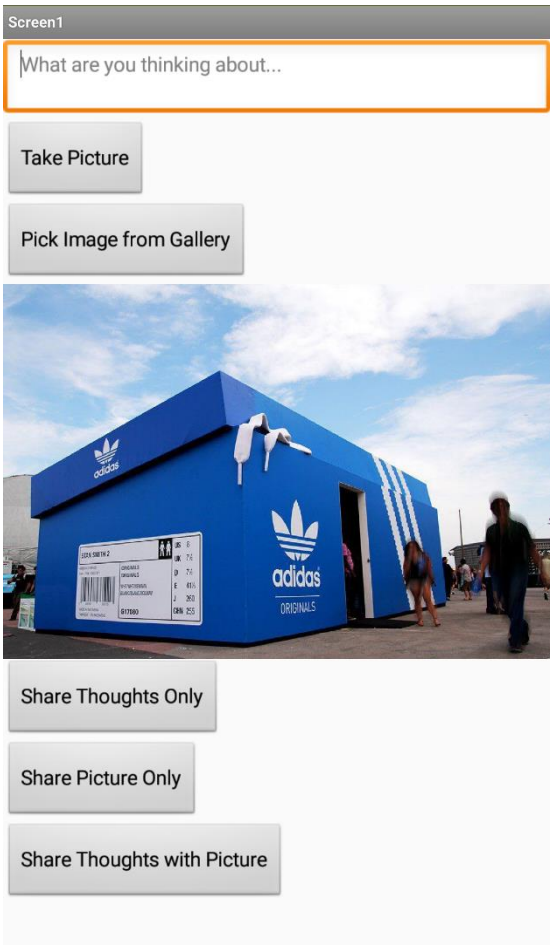
The ImagePicker's 'Open' command block instructs the component to perform its main function, which is to open the device's image gallery and allow the user to select an image from it. This block is automatically executed when the button displayed by the component in the UI, a button is clicked.

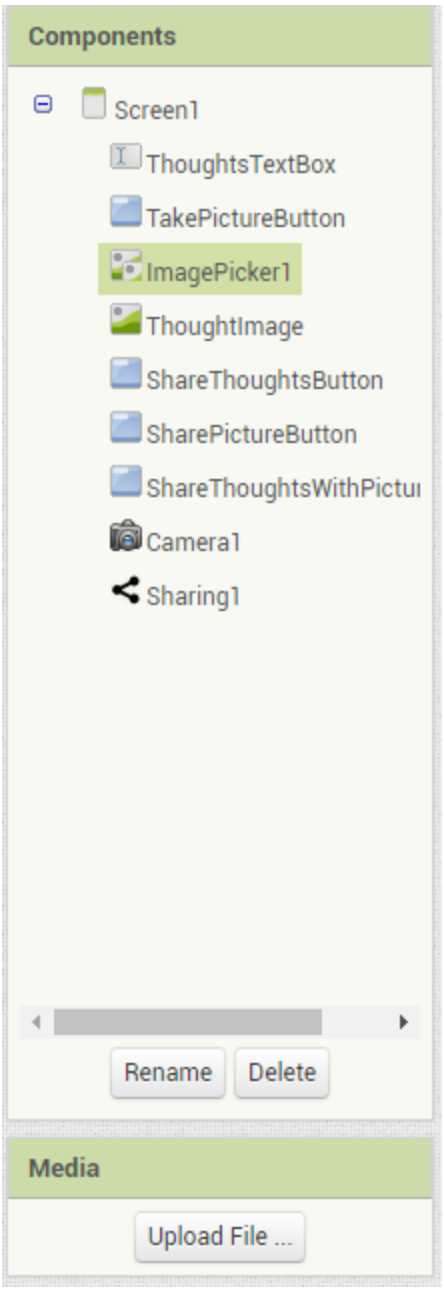
The 'AfterPicking' event handler



The ImagePicker component's 'AfterPicking' event is triggered after the user selects an image from the image gallery opened when the component's UI element, a button, is clicked. Note that the component does not provide a 'WhenPicked' event handler, because the only action that should occur when the user picks an image is to set the 'Selection' property of the component to the image that was picked, and that is done automatically by the component and does not require manual programming.

We can use the ImagePicker component to improve the user experience on MyThoughts, so that the user can pick an image from the device gallery, as well as the camera, to share with other apps.


Screen1
What are you thinking about...
Take Picture
Pick Image from Gallery
adidas
ORIGINALS
Share Thoughts Only
Share Picture Only
Share Thoughts with Picture

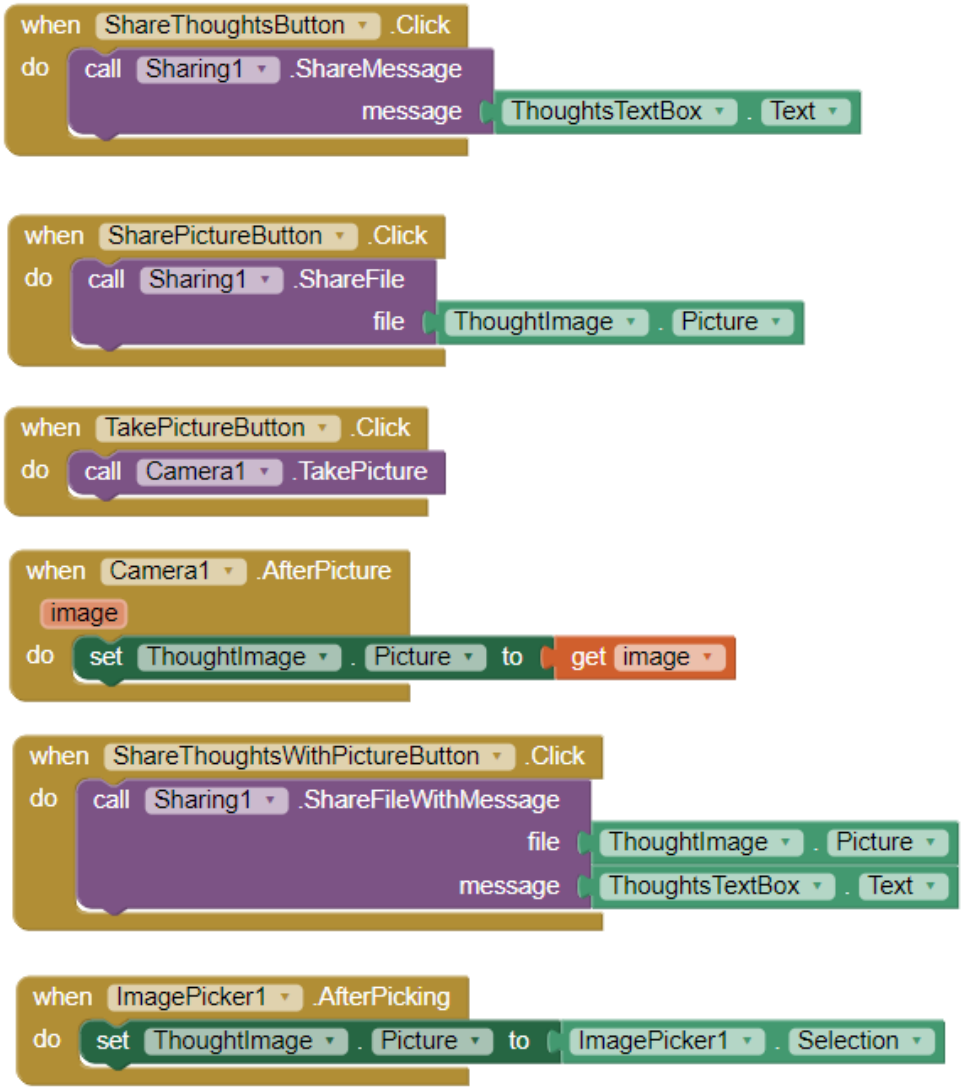


Updated components list

Program your app as follows:

- When the user selects an image through the ImagePicker component, set the 'Picture' property of ThoughtImage (the Image component) to the *Selected* property of the ImagePicker component. This should make the Image component display the image picked by the user from the image gallery in the app, like the app screenshot above.

Here is the entire code for the complete MyThoughts app:



Summary

The following are the key points covered in this module:

- The Sharing component gives our Android apps the ability to share messages and files with other apps on an Android device.
- When any of the Sharing component's Share* command blocks are executed, the Android device displays a list of installed apps that can handle the type of data being shared.

- The Camera component uses a device's camera to capture images for use in our Android apps. It is instructed to perform its main function through the *TakePicture* command block, and its *AfterPicture* event block provides the image captured as an event parameter.
- The Image component is a visual component whose primary function in App Inventor is to display images. The image it displays is contained in its Picture property.
- The ImagePicker component displays a button, which when clicked, allows the user to pick images from the device's image gallery for use in our Android apps. The image selected by the user is stored in the ImagePicker's 'Selection' property which is not available in the component's property section, and is only accessible through a property getter block.
- It is important to enhance the user experience with your apps as much as possible with the available components in App Inventor.

MODULE 6

Introduction to App Memory & Decision-Making

Table of Contents

Learning Objectives

By the end of this module, the student would be able to:

- Program Android apps to store information in memory using variables.
- Program Android apps to make decisions and alter the flow of execution of a program.
- Perform comparisons and Boolean logic with relational and logical operators.
- Display popup dialogs in an Android app.
- Progressively add features to Android apps to make the app more sophisticated.

Module Prerequisites

- App Inventor emulator or a physical Android device.
- A reliable internet connection should be made available on the student’s laptop/computer, and if using an Android device for app testing instead of the emulator, both the computer and the Android device should be connected to the same Wi-Fi network.
- A Google (Gmail) account is needed by the student to create projects and make Android apps on the App Inventor platform. Login details should either be provided to the student, or the login should be done for the student when needed.

Programming an App with Memory

Just like human beings, an application can remember things. In fact, the ability of an app to remember information is essential to the basic running of its operations, and we have seen this demonstrated in the apps we have built so far with App Inventor; an app remembers the values of the properties of every component added to it in the Designer Window during development, and still keeps these values in memory when the app is run on an emulator or Android device for testing, even if the values are changed with property setter blocks in reaction to events.

App memory can be visualised as a storage room that can hold any type of information valid in App Inventor.

[image of stack of lockers, like below]

When you add a new component to an app, App Inventor adds a new row for the component, and adds a locker for every property of the component to the component's row in the storage room.

[image of just one row of the lockers in the previous image]

Each row is tagged with the component's name, each locker is tagged with the property name, and the content of each locker is the value of the property. Every component comes with an initial set of property values, so each locker would have some content right from when it was created and added.

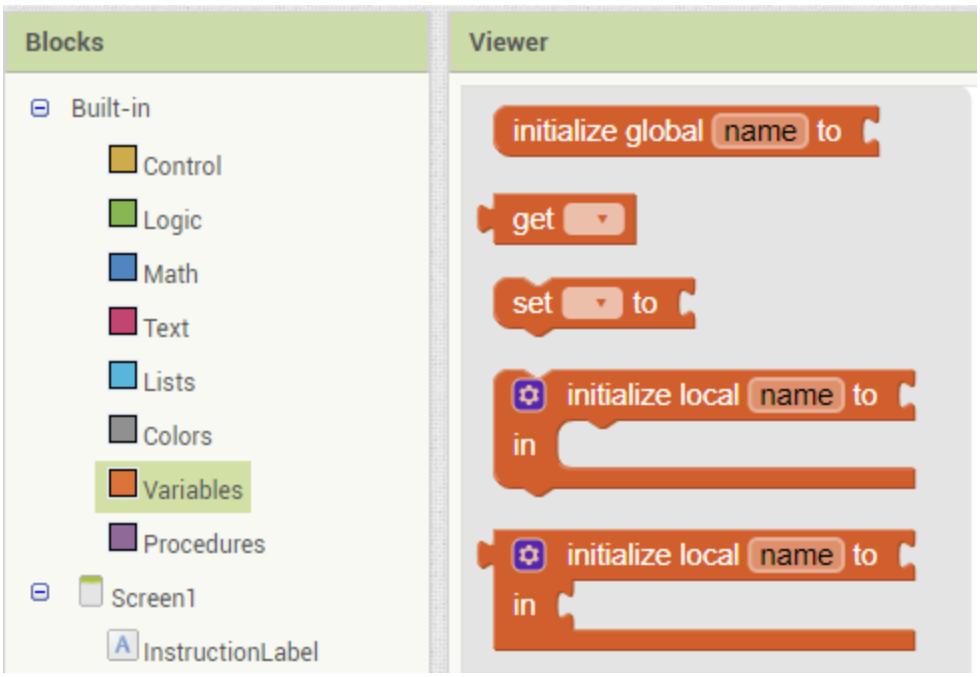
Executing a property getter block is the equivalent of walking to the property locker and retrieving its contents. Similarly, executing a property setter block is the equivalent of walking to the property locker and replacing its contents. You know which locker to go to because the property setter/getter block gives you a name, which matches the name with which a property locker is tagged.

In this way, an app keeps track of the properties of components in its memory, and it is all done automatically. That is, you do not need to program the app to create memory slots with tags for a component's properties; this is done automatically when you add a component to the app.

However, there are situations when you would need your app to keep some information that is not specific to a component in its memory. If you were building a game app, for example, you might want to keep track of how many times the user has performed a specific action and display it only after the game has ended. If you were to display this number throughout the game, it would be suitable to store the number in the Text property of a Label component. However, because you only need to display it at **the end** of the game, you need another way to program your app to keep this number in its memory, as there is no suitable component property to store it in. App Inventor allows you to do this with **Variables**.

Introduction to Variables

Variables are used to store information that is not tied to a particular component in an app. Like a component property, a variable has a name by which it is referred to, and a value which represents the information it was created to store. The *Variables* built-in block category in the Blocks Editor provides blocks for creating and managing variables in App Inventor.



Creating a Variable

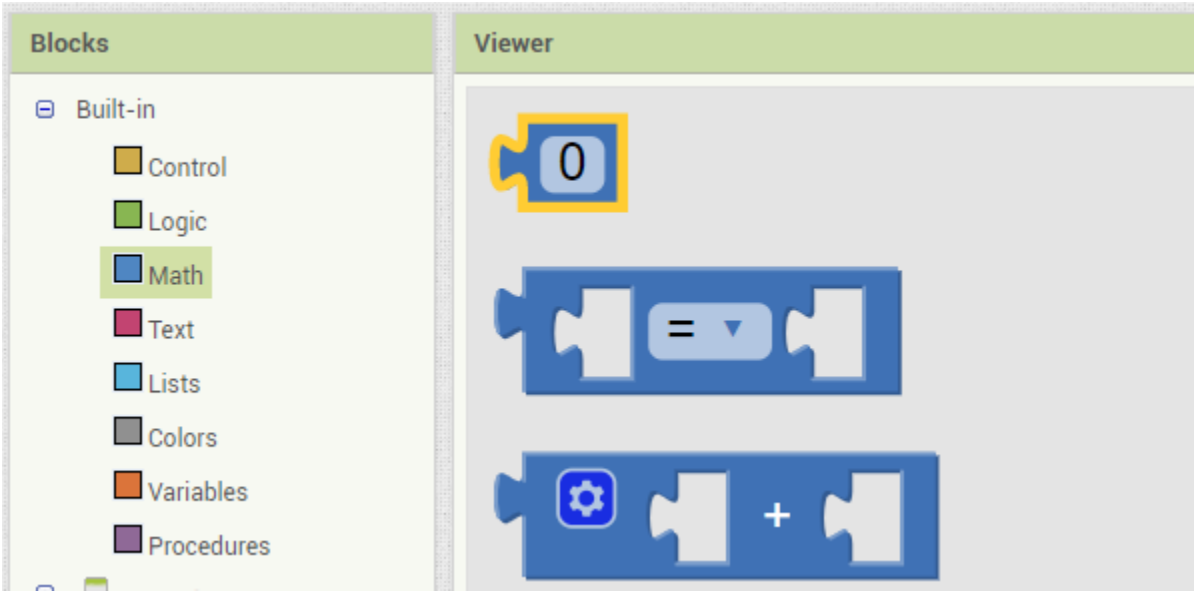
To create a variable, drag the *initialize global name to* block from the *Variables* built-in block category to the Viewer.

Like components, every variable is given a default name (the variable block above has the name 'name'). A variable must have a unique name, that is, a name that is not possessed by any other variable in your app, so it is a good practice to rename a variable to indicate what it is used for in the app at the point of creation.

INFO: Drag another *initialize global name to* block to the Viewer and check its name; it is given the name 'name2' because the first variable you created already has the default name, 'name' and variable names must be unique.

To rename a variable, click on the 'name' field and edit it to your desired value.

The block says ‘initialize’, which means a variable must be initialized to a value when it is created; that is, a variable must be given an initial value when it is created. This is similar to the way every component property is given an initial value when the component is added to the app. A variable can be given any type of value; to initialize the score variable above to a value of 0 (a number), drag the number block from the *Math* built-in block category and snap it to the variable block.



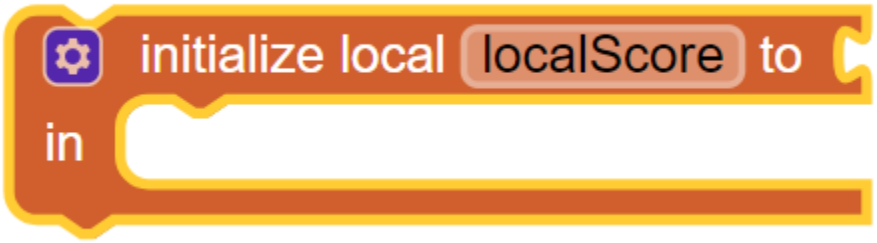
Using and Modifying a Variable

When a variable is created and initialized as we have done above, two blocks are created for it - a getter block, and a setter block. Just as it is in component properties, these blocks can be used to retrieve the value of a variable for use in another part of our code, and to modify the value of a variable from any part of our code. To see a variable’s blocks, hover over the name field of the variable block.

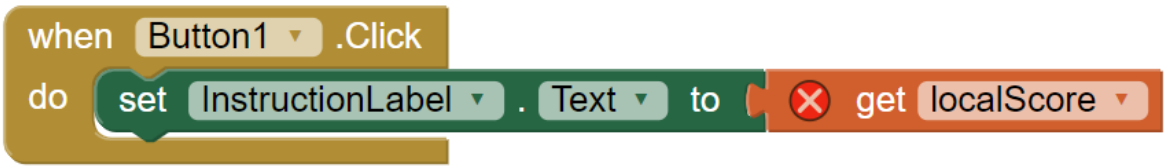
You can then drag these blocks and use them in any part of your code where it is needed.

Global & Local Variables

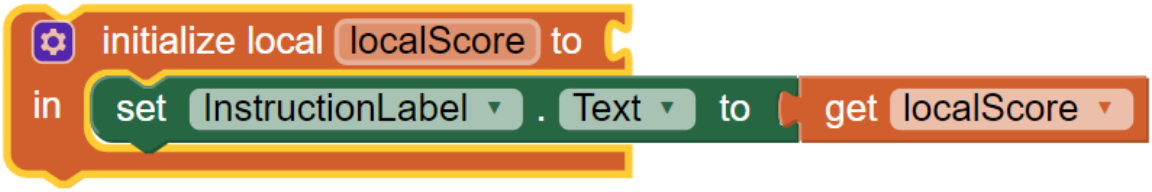
The variable blocks we have been using to create and manipulate variables so far all have the word "global" on them. This means that they can be accessed and modified from any part of your code. App Inventor also allows you to create local variables with blocks from the *Variables* built-in block category.



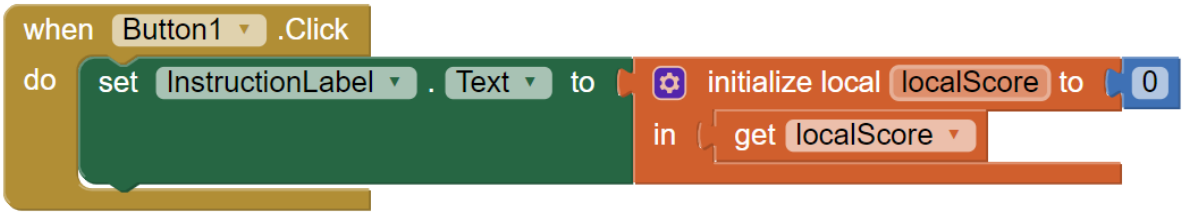
The block above is a form of control block that can be stacked above or below other blocks, and can contain other blocks in it. The variable that is created with this block is local to the blocks within it; that is, the variable can only be seen (accessed and modified) by the blocks enclosed within it. In this case, the following block would not work:



But the following would work:



Local variables can also be created in the form of reporter blocks, in which case the variable would only be accessible to the blocks into which the variable reporter block is stacked.



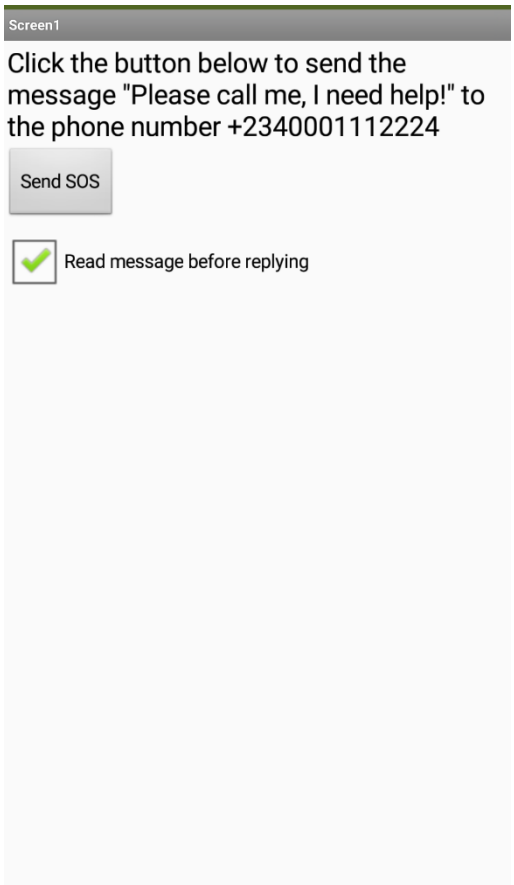
the local variable named 'localScore' is only accessible to the *set InstructionLabel.Text* block

As a programmer, being able to program an app's memory opens up a whole lot of possibilities, as regards the functionality that can be provided in an app. With the ability to remember things, comes the ability to make decisions. An application is also capable of making decisions based on the state and condition of certain component properties and variables contained in it.

Programming an App to Make Decisions

When blocks were introduced in this textbook, it was said that an app executes blocks in the order in which they are stacked in the Blocks Editor. This is the default flow of a program; a top-to-bottom execution order which is easy to follow when building basic apps. As apps get bigger and more sophisticated, however, there comes a time when it is necessary to alter that natural flow based on certain conditions.

It is common for apps to offer the user the ability to control the app's operation with settings. For example, the MessageReader app we built in the fourth module could give the user the ability to tick a checkbox if they want the app to read received messages before sending the automated reply, or untick it if otherwise.



With this added feature, the app would have to decide whether to read the message out loud or not, depending on whether the checkbox is ticked or not.

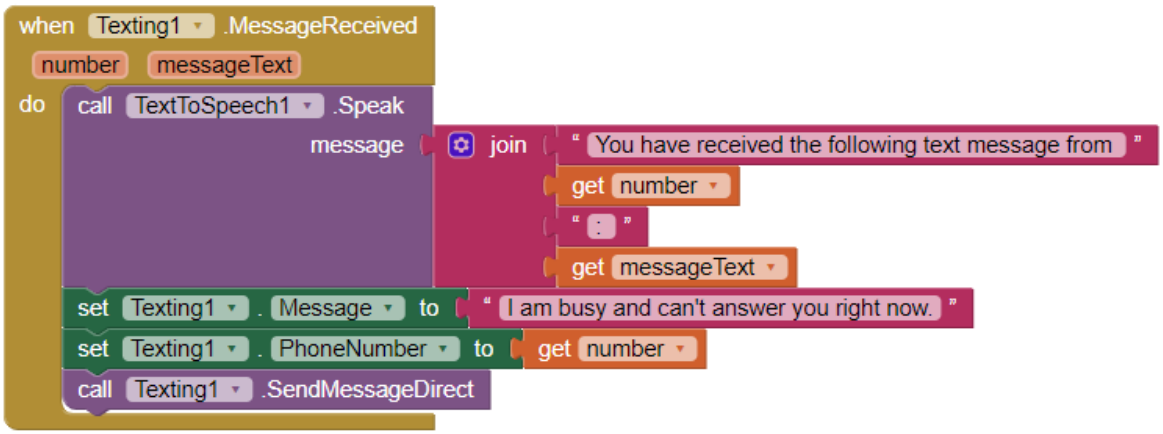
App Inventor provides a set of blocks for programming decision-making into apps in the *Control* built-in blocks category of the Blocks Editor.

The If-Then Block

Decision-making consists of two parts: a condition that is tested to decide whether a course of action should be followed, and an action or set of actions to be performed depending on whether the condition is valid or not. The first part is about asking questions, and the second part is about acting based on the answer(s) to the question(s) asked. When questions are being asked for the purpose of making decisions, it is a common thing to use the words “if” and “then”.

For example, let us analyze, in words, what would happen in the MessageReader app if the behaviour described in the previous section was to be programmed into it.

These are the programming blocks for the app prior to the addition of decision-making:



The app reads the message and then sends a reply to the sender of the text message

When a message is received, the app has to decide whether to read the message aloud or not, depending on whether the user ticked the checkbox or not.

The first thing to be done when a decision has to be made is to define a condition to test for. The condition, in this case, is

- **Condition:** the state of the checkbox (ticked or not ticked).

Then, the app has to define an action to be taken if the condition is passed - in this case, if the checkbox is ticked. This step is compulsory. For the MessageReader app, this action would be:

- **Action A:** read the message.

The app can optionally define an action to be taken if the condition is not passed. This step is not compulsory. For the MessageReader app, this action would be:

- **Action B:** do not read the message.

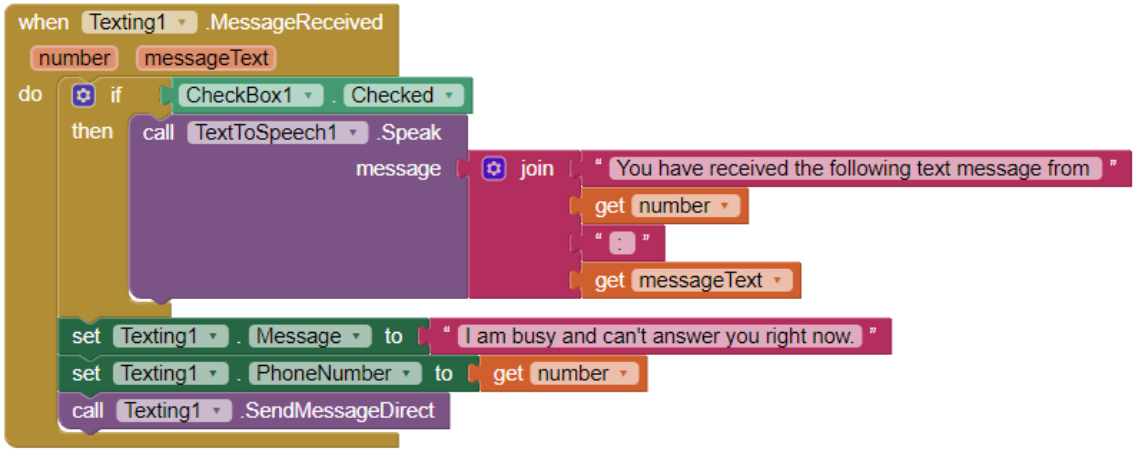
The above steps are taken when the app is being programmed. When the time comes for the app to actually make the decision, the following statements would be made:

- **Statement A:** Did the user tick the checkbox?
  Whatever answer is given to this question can be reduced to the form **true** or **false**. This type of answer is called a Boolean value, and it is essential in decision-making, as it is the result of testing a condition that determines what actions are performed next.

- **Statement B:** If the user ticked the checkbox, perform Action A.
  This can also be written as: "If Statement A is true, perform Action A".
- **Statement C:** If the user did not tick the checkbox, perform Action B.
  This can also be written as: "If Statement A is false, perform Action B".
- The app then goes on to execute the rest of the blocks in the normal order, which is from top to bottom.

Since Action B is the exact opposite of Action A, there is no need to define it when programming the app, and Statement C is unnecessary.

The programming blocks for the app would now look like this:



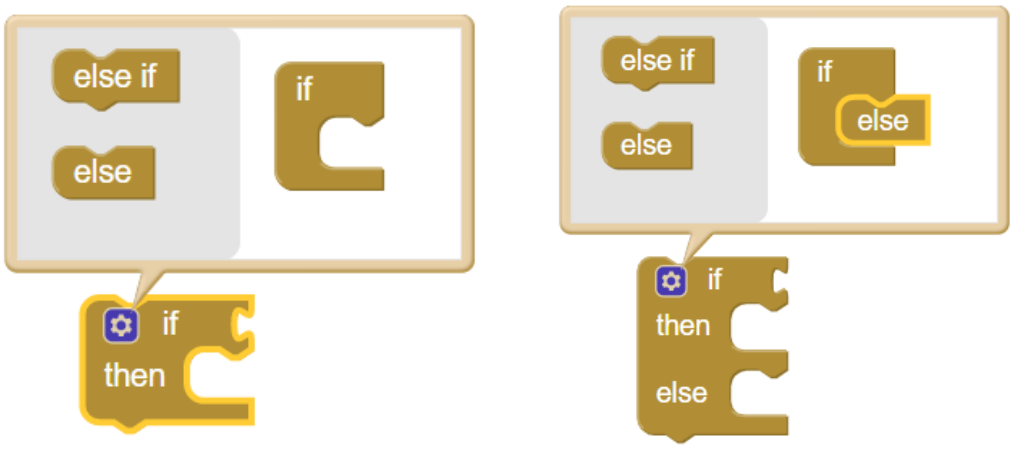
The script above first tests the condition "is the checkbox checked (ticked)?" using the property reporter block of the Checkbox component. If the answer is true, it reads the text message aloud, and then continues its regular operation of sending a reply back to the sender of the text message. If the answer is false, the app does not read the text message aloud, but still goes on to execute the blocks for sending the reply to the sender, since these blocks are defined outside the condition.

The block used for the decision-making act above is a control block called a "If-Then" block.

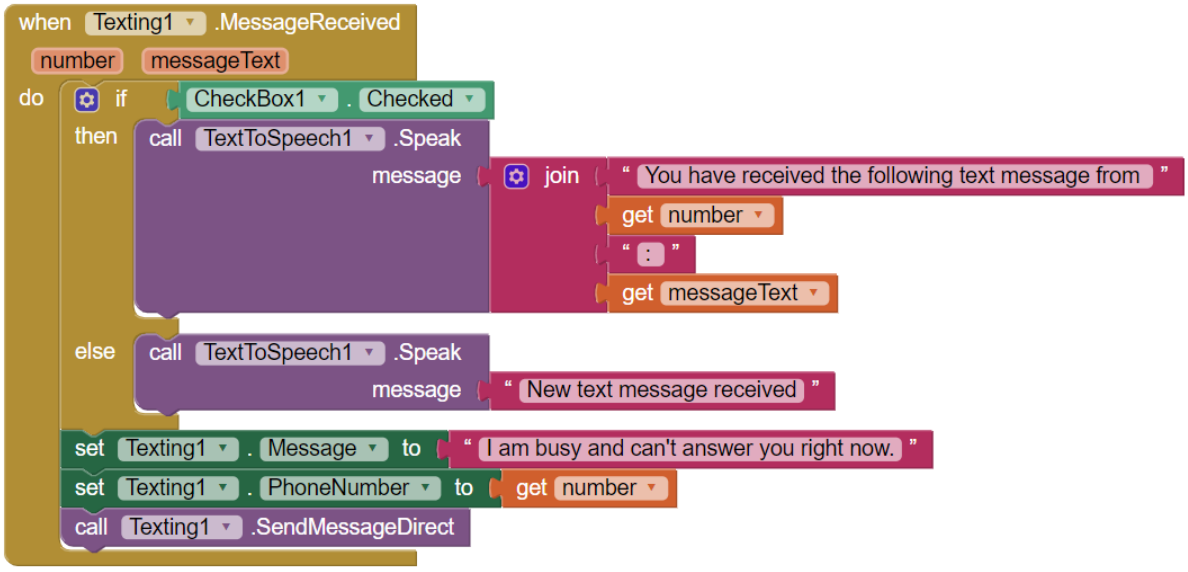
The block consists of two sections: an "if" section and a "then" section. The "if" section is where the testing of the condition takes place, and it, therefore, expects a reporter block which reports **true** or **false**. The "then" section is where the blocks for the action to be performed if the condition is passed (that is, if it is true) are placed.

"Else" and "Else-If"

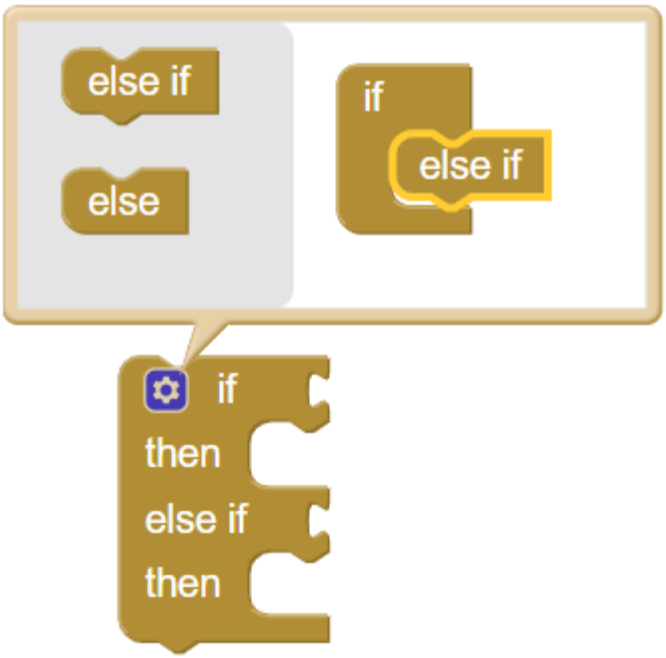
A third section can be added to the If-Then block, in which the blocks for the action to be performed if the condition tested in the "if" section is not passed. This section is called "else" and can be added by clicking the settings icon at the top left of the If-Then block and dragging an else block into the *if* section.



For example, if we wanted the MessageReader app to just say “New text message received” instead of reading the actual message when the checkbox is unticked, we would arrange the blocks as follows:

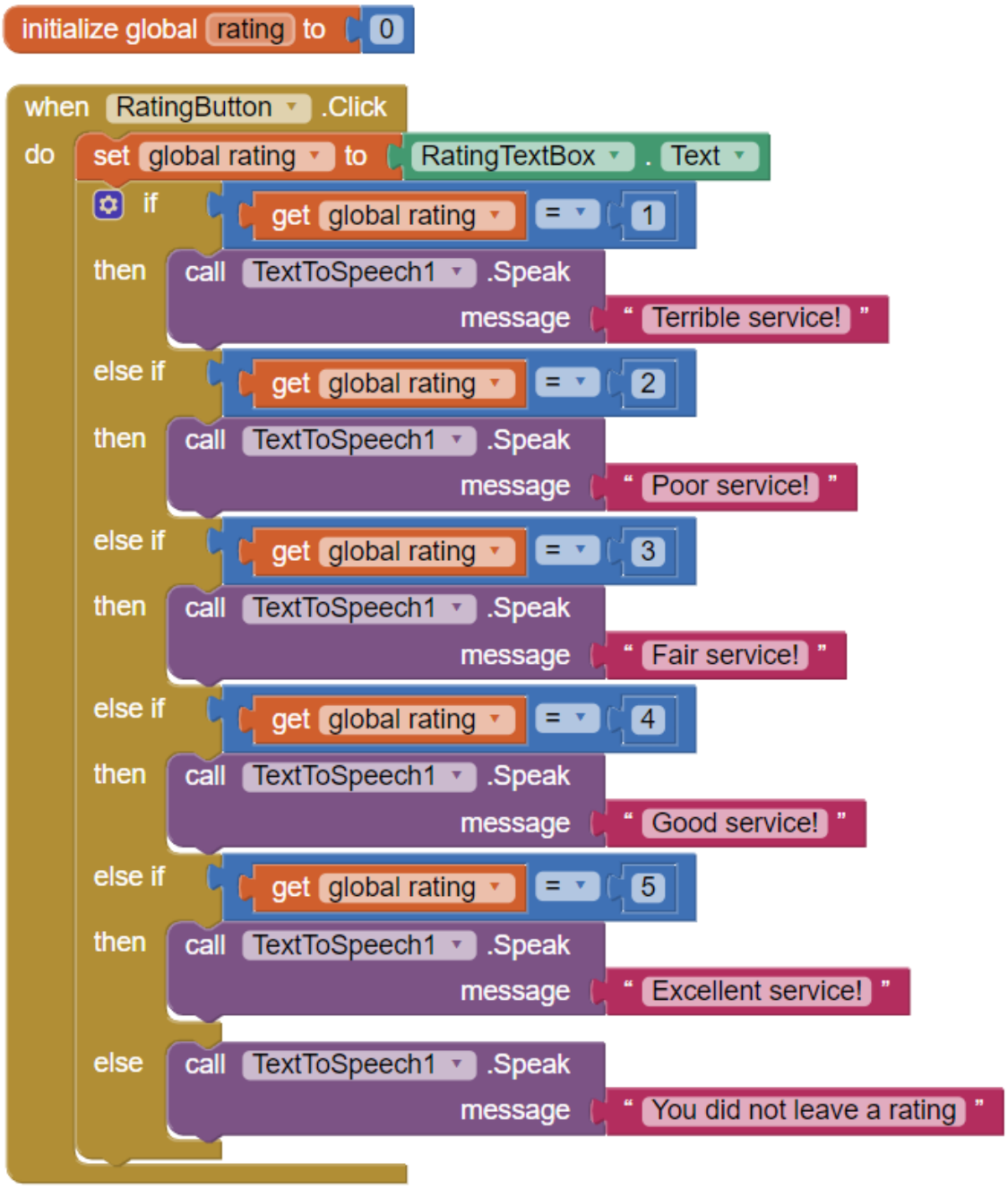


Often when making decisions, we have to do certain things based on more than one condition. App Inventor also allows that with the “else-if” block which can be used to test for another condition if the condition in the *if* block above it is not passed. The *else-if* section can be added to an If-Then block by clicking the settings icon at the top left of the If-Then block and dragging an else block into the *if* section.



Like the If-Then block, the *else-if* block has an “if” section for testing conditions, and a “then” section for specifying actions to be performed, that is the blocks to be executed, if the condition tested for in the “if” section is passed.

An example of where an *else-if* block would be useful, is if we wanted to perform different actions based on different values of a property or variable. Users of a service are often asked to rate the service between 1 (for terrible service) and 5 (for excellent service) after using the service. We could program an app to allow the user of our service to type in a number to rate the service, and then click a button to hear the rating in words as follows:

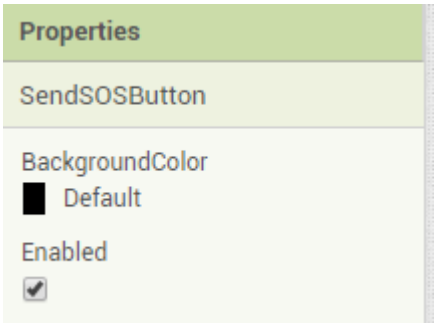


The first block creates and initializes a variable named *rating* to 0. When the button is clicked, the variable is set to the value entered by the user in a TextBox. The app then tests and speaks out different things depending on the value of the *rating* variable, using if-then and else-if blocks. The last *else* block does not have if-then sections, and the block in it is executed only if the other if-then and else-if blocks above it do not pass their condition tests.

An if-then block can have as many else-if blocks as are required for the conditions that need to be tested, but there can be just one *else* block. When an *else* block is present, the blocks enclosed in it are executed only if none of the conditions for the if-then and else-if blocks above it was passed.

Boolean Properties, Variables & Expressions

It was mentioned earlier that anything that can be reduced to the form **true** or **false** is called a Boolean value. Many components properties have boolean values; for example, a Button component has an *Enabled* property whose value is determined by a checkbox:



When this box is checked, the value of the *Enabled* property becomes **true**, which means that the button component is enabled. On the other hand, when this box is unchecked, the value of the *Enabled* property becomes **false**, which means that the button component is disabled (not enabled).

TRY IT OUT: Check and uncheck the *Enabled* property of a button component while live testing it on an emulator or Android device through the AI2 Companion app. Observe the changes made to the button while doing this.

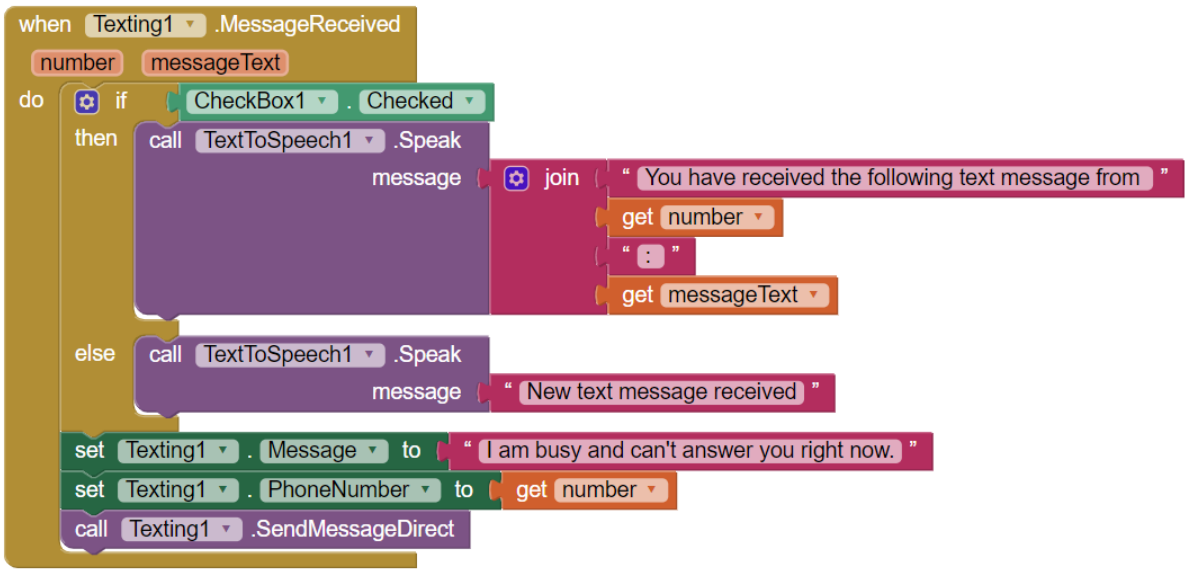
This *Enabled* property has a corresponding property getter block which returns true if the box is checked, and false if otherwise.

This type of property whose value is a Boolean value, is called a Boolean property.

We have seen one of such properties in use, in form of the *Checked* property of the CheckBox component.

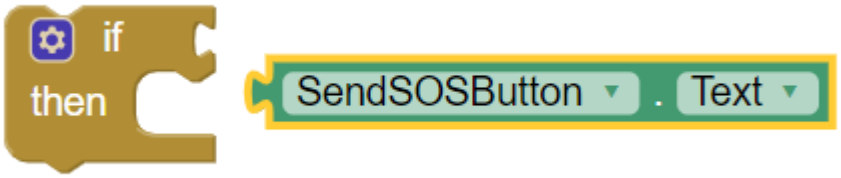
The value of a variable can also be set to true or false, making it a Boolean variable.

Because the *CheckBox1.Checked* and *get global isCheckBoxChecked* blocks directly return Boolean values, they can be directly used as the condition in an *if* block.



Suppose, however, that we want to make decisions based on whether a variable that contains a number value, like the score variable created earlier, is equal to a particular number, say 1 or 20, or whether the *Text* property of a Button component is equal to the text “Send SOS”. In these cases, we cannot directly use these blocks as the condition in an *if* block.

This block does not work for testing if the score variable is equal to 1, or 20, or any other number.

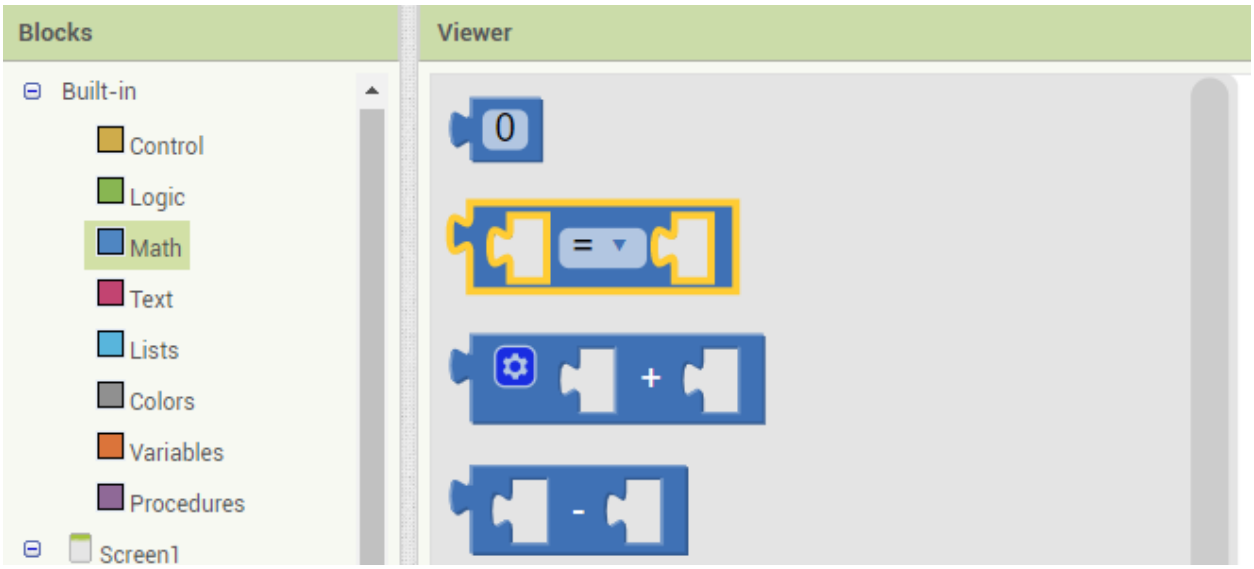


In this case, the blocks do not even snap together because the property getter block is not a valid Boolean value.

In cases like this, you need a way to test conditions by comparing two or more variables, properties or values and return the result of the comparison as a Boolean. App Inventor makes this possible with relational and logical blocks.

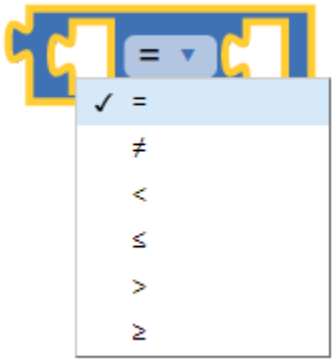
Testing Conditions with Relational & Logical Blocks

Relational blocks are used to compare the relation between two blocks. There are different types of relational blocks, depending on the operation you want to perform. These blocks are found in the *Math* built-in block category.



operation: equal to

A relational block takes two inputs. The block above compares the two inputs and returns **true** if they are equal, or **false** if not. To perform an operation other than 'equals (=)', click on the '=' sign and select the desired operation.



operation: not equal to

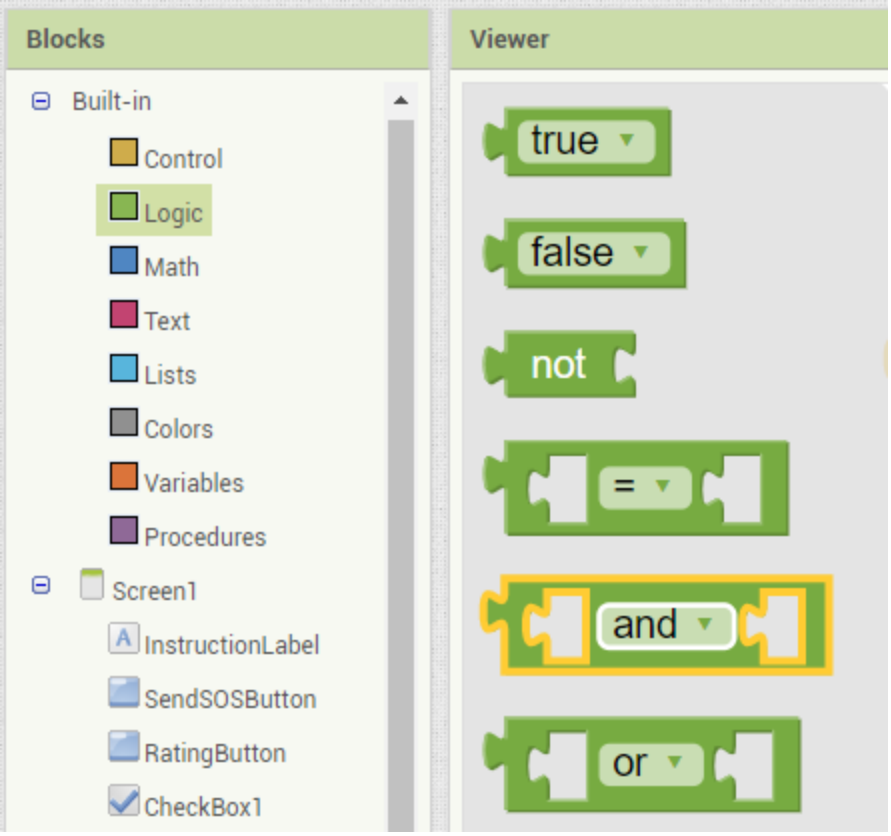
operation: less than

operation: less than or equal to

operation: greater than

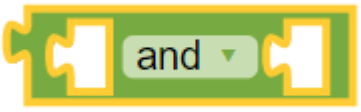
operation: greater than or equal to

Logical operators are used to connect two boolean values, or two relational blocks together. They also return either true or false, depending on the operation being performed. Logical blocks can be found in the *Logic* built-in block category.

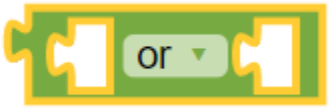


the last three are the logical blocks used for testing conditions.

this block works similarly to the *equals* relational block



this block returns true only if both inputs are true, and false if any of the inputs is false



this block returns true if any of the inputs are true, or false if both inputs are false

Any block that makes some computations and returns a Boolean value, is called a **Boolean expression**.

The combination of variables, If-Then, Else-If, Else, Relational and Logical blocks gives a programmer the ability to create apps that perform sophisticated functions. Using the knowledge gained so far in this module, we will build an app that calculates the average score and grade of the scores of three of a student's tests in the next module.

Summary

- An application can keep information in its memory.
- Memory slots are automatically created in an app for component properties with initial values when components are added to an app.
- An application can be programmed to store information not tied to components with variables.
- Like component properties, a variable must have a name and an initial value.
- When a variable is initialized, getter and setter blocks are created for it.
- A variable can be initialized as global to make it available for use in every part of an app's code, or local to limit its access to specific sections of an app's code.
- An application can be programmed to make decisions with blocks (If-Then, Else-If, Else) from the *Control* built-in blocks category.
- Decision-making consists of testing if a condition is **true** or **false**, and performing actions based on the result of the testing.
- Anything that can be reduced to either **true** or **false** is called a Boolean.
- A type of property whose value is a Boolean value, is called a Boolean property.
- A variable whose value is set to true or false is a Boolean variable.
- Any block that makes some computations and returns a Boolean value is called a **Boolean expression**.
- Boolean expressions are made using relational and logical blocks which compare two or more values.

MODULE 7

Building the TestAverage App

Table of Contents

Learning Objectives

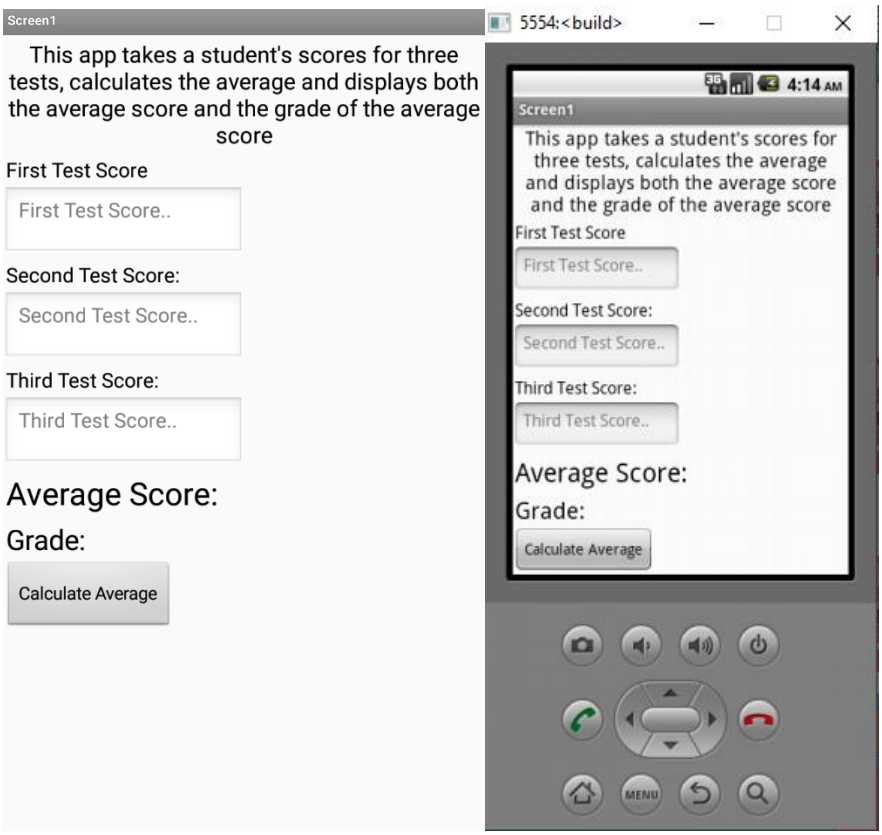
By the end of this module, the student would be able to:

- Display popup dialogs in an Android app.
- Progressively add features to Android apps to make the app more sophisticated.
- Build the TestAverage application.

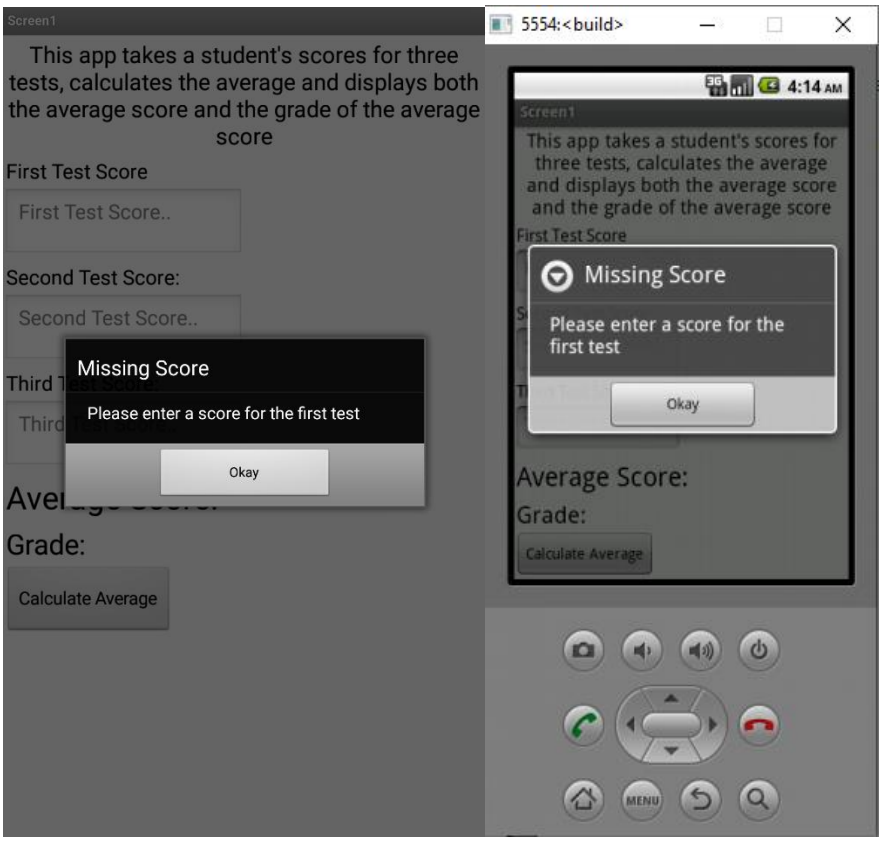
Module Prerequisites

- App Inventor emulator or a physical Android device.
- A reliable internet connection should be made available on the student's laptop/computer, and if using an Android device for app testing instead of the emulator, both the computer and the Android device should be connected to the same Wi-Fi network.
- A Google (Gmail) account is needed by the student to create projects and make Android apps on the App Inventor platform. Login details should either be provided to the student, or the login should be done for the student when needed.

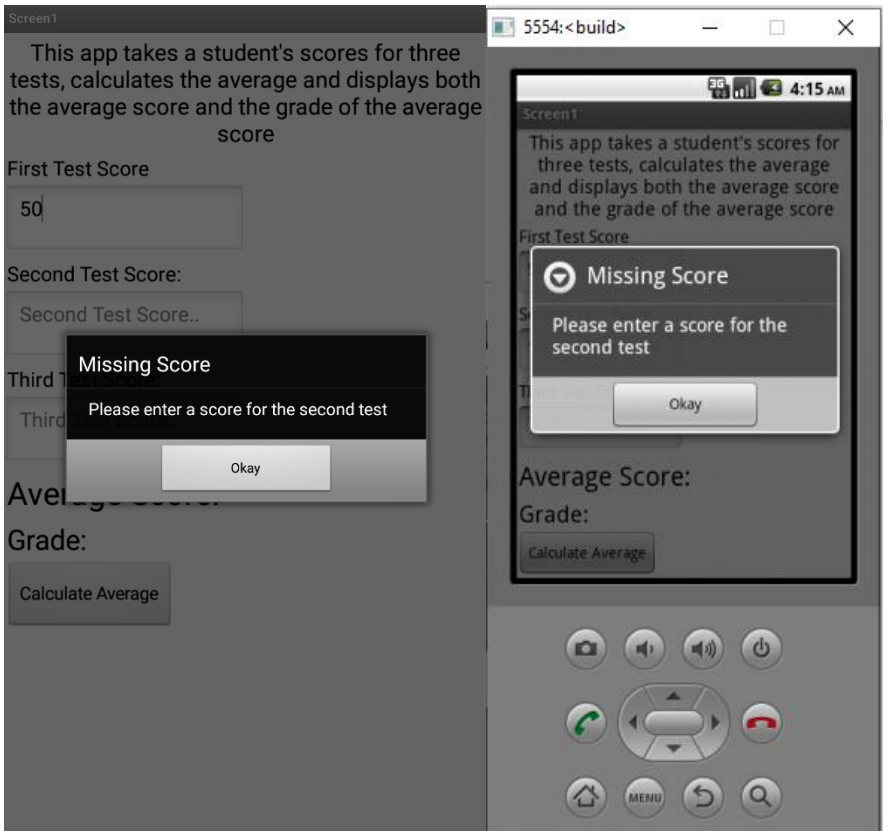
The TestAverage App

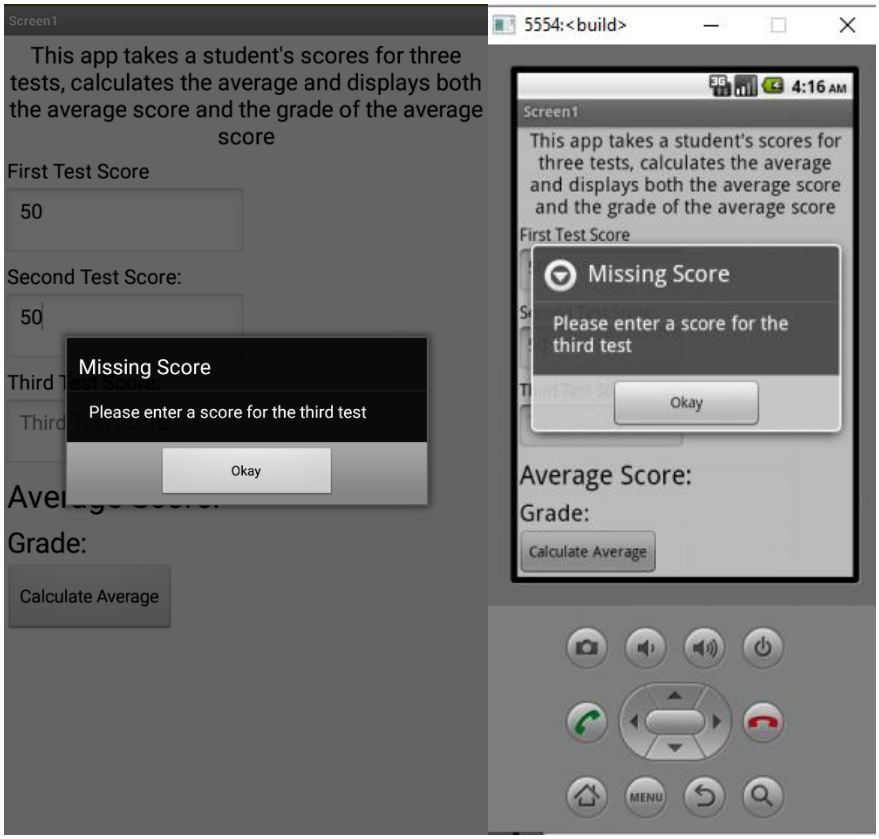


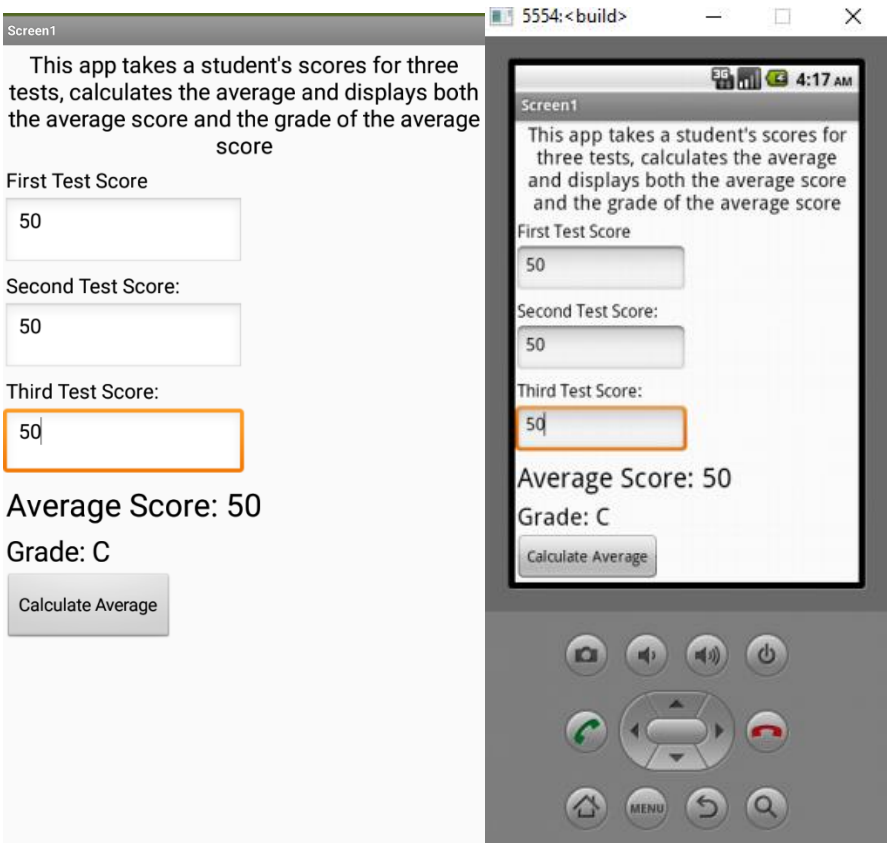
The app allows the user to enter three test scores, and then click on the “Calculate Average” button to calculate and display the average score and grade.



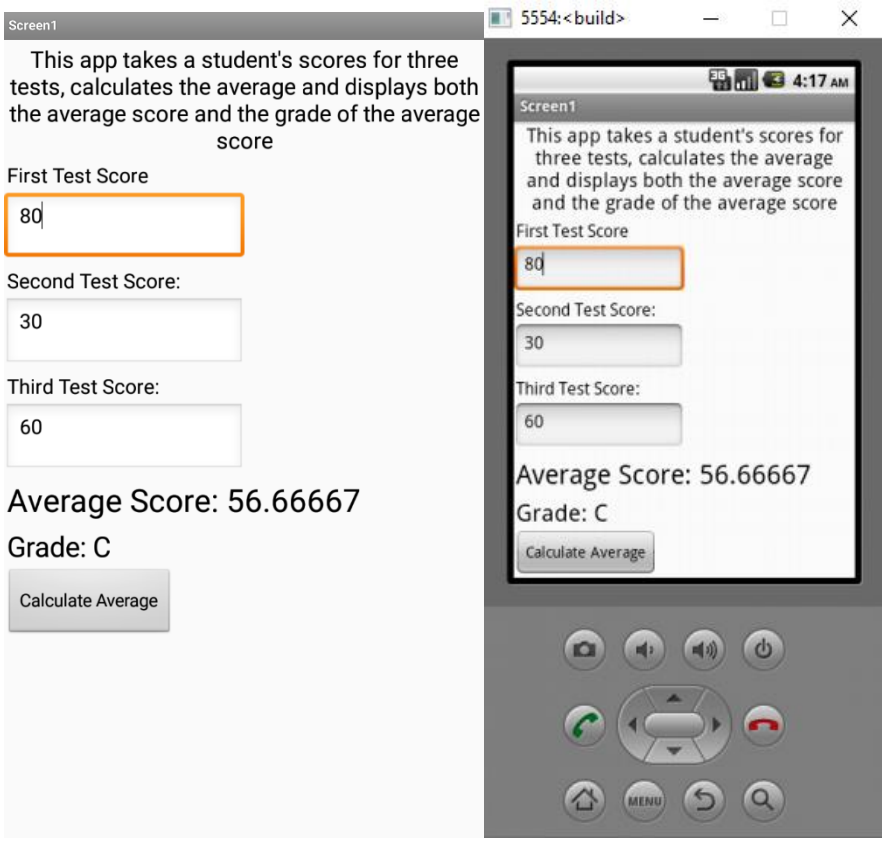
If the user clicks on the “Calculate Average” button without filling in a test score, the app shows a popup dialog that informs the user of this. The dialog disappears when the user clicks on the “okay” button.


Screen1
This app takes a student's scores for three tests, calculates the average and displays both the average score and the grade of the average score
First Test Score
50
Second Test Score:
Second Test Score..
Missing Score
Please enter a score for the second test
Okay
Grade:
Calculate Average
5554:<build>
4:15 AM
Screen1
This app takes a student's scores for three tests, calculates the average and displays both the average score and the grade of the average score
First Test Score
Missing Score
Please enter a score for the second test
Okay
Average Score:
Grade:
Calculate Average
MENU


Screen1
This app takes a student's scores for three tests, calculates the average and displays both the average score and the grade of the average score
First Test Score
50
Second Test Score:
50
Missing Score
Please enter a score for the third test
Okay
Grade:
Calculate Average
5554:<build>
4:16 AM
Screen1
This app takes a student's scores for three tests, calculates the average and displays both the average score and the grade of the average score
First Test Score
Missing Score
Please enter a score for the third test
Okay
Average Score:
Grade:
Calculate Average
MENU



When the user enters scores for all three tests and clicks on the “Calculate Average” button, the average score and grade are displayed

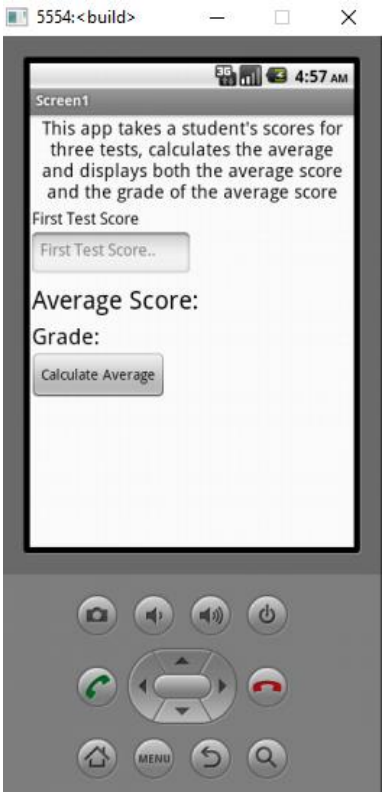


This application looks quite complex (and it is), but it would be fun to build, thanks to variables, control blocks and boolean operations. We will build it bit by bit, starting from calculating the average for one test score, to two, and finally for three test scores.

Create a new App Inventor project and name it “TestAverage”.

Designing the User Interface

We will start with the following user interface:



Design the UI with the following components and properties:

COMPONENT TYPE	COMPONENT NAME	COMPONENT PROPERTIES
Label	InstructionLabel	- Text: “This app takes a student's scores for three tests, calculates the average and displays both the average score and the grade of the average score” - FontSize: 18 - TextAlignment: center
Label	FirstTestScoreLabel	- Text: “First Test Score” - FontSize: 16
TextBox	FirstTextScoreTextBox	- Hint: “First Test Score…” - FontSize: 16
Label	AverageScoreLabel	- Text: “Average Score:”

		- FontSize: 25
Label	GradeLabel	- Text: “Grade:” - FontSize: 22
Button	CalculateAverageButton	Text: “Calculate Average”

Since you are already familiar with the process of designing a UI, we will be creating a table like this for our module projects going forward.

Programming the App’s Behaviour

When you are done designing the UI with the components above, switch to the blocks editor to program the components.

Since we have learned about variables, the first thing to do before defining app behaviour would be to create and initialize every global variable that would be needed in the app. At this point, we know that we would need one variable to hold the first test score, so let us create that:

The TestAverage app in its current state, with one test score option, does not make sense since the average of one value is that value, but we are building the app this way in order to establish basic building blocks that would make it easier to program the app as we add more test score options and things get more complex. For now, we would assume that the average of a value is that value divided by 3.

For grade calculations, we will use the following table:

70-100	A
60-69	B
50-59	C
45-49	D
40-44	E

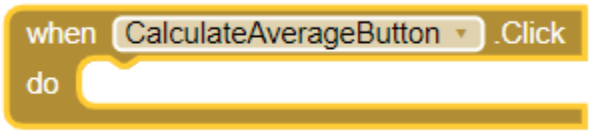
0-39	F

We need the following behaviours in the TestAverage app:

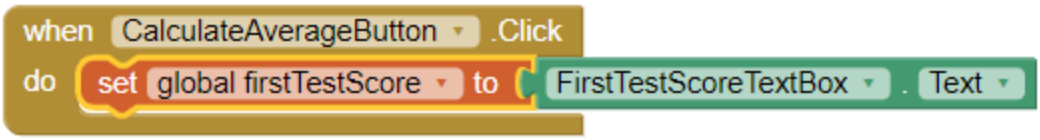
- When *CalculateAverageButton* is clicked
    - Set the value of *firstTestScore* to the *Text* property of *FirstTestScoreTextBox*.
    - Test the value of *firstTestScore*:
        - If it is empty, show a popup dialog notifying the user to enter a score for the first test.
        - Else, go on to calculate the average as follows:
            - Divide *firstTestScore* by 3.
            - Create a local variable *averageScore* and initialize it to the result of the division above.
            - Inside *averageScore's* local variable block,
                - Set the *Text* property of *AverageScoreLabel* to "Average Score: *averageScore*".
                - Test the value of *averageScore*:
                    - If it is greater than or equal to 70, set the *Text* property of *GradeLabel* to "A".
                    - Else, if it is greater than or equal to 60, set the *Text* property of *GradeLabel* to "B".
                    - Else, if it is greater than or equal to 50, set the *Text* property of *GradeLabel* to "C".
                    - Else, if it is greater than or equal to 45, set the *Text* property of *GradeLabel* to "D".
                    - Else, if it is greater than or equal to 40, set the *Text* property of *GradeLabel* to "E".
                    - Else, set the *Text* property of *GradeLabel* to "F".

We will tackle these behaviours one after the other.

1. When *CalculateAverageButton* is clicked

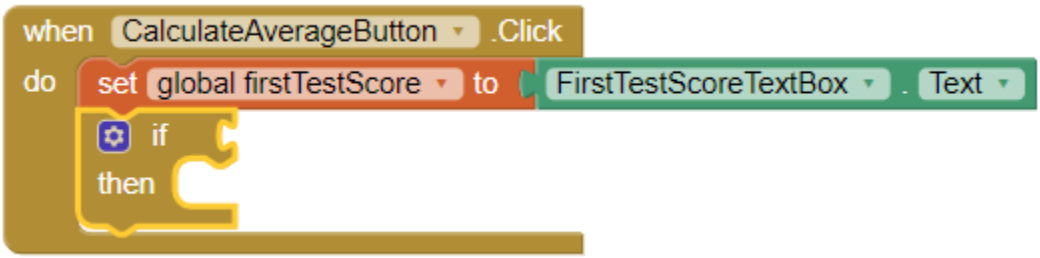


2. Set the value of *firstTestScore* to the *Text* property of *FirstTestScoreTextBox*.

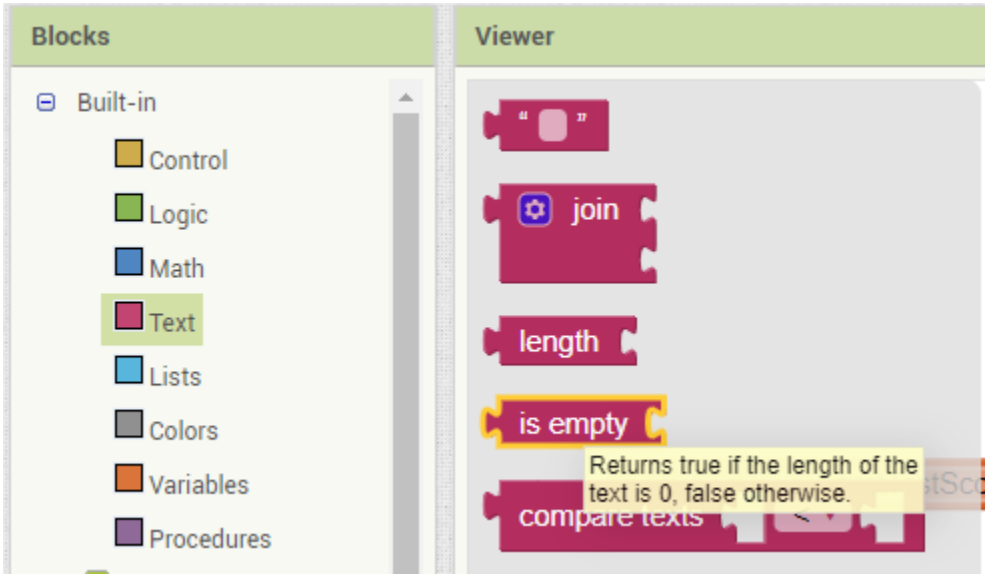


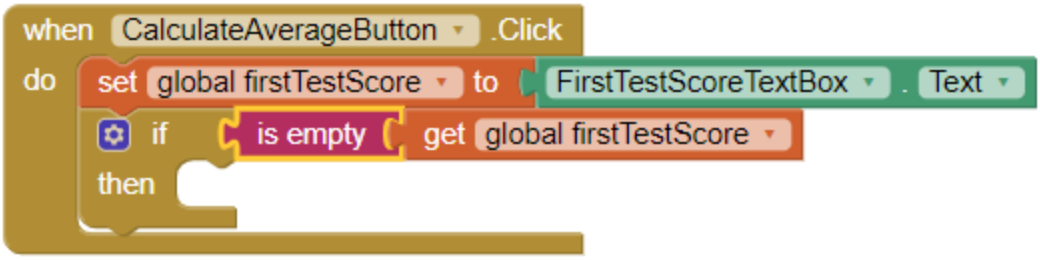
3. Test the value of *firstTestScore*: If it is empty, show a popup dialog notifying the user to enter a score for the first test.

We start by adding an If-Then block.



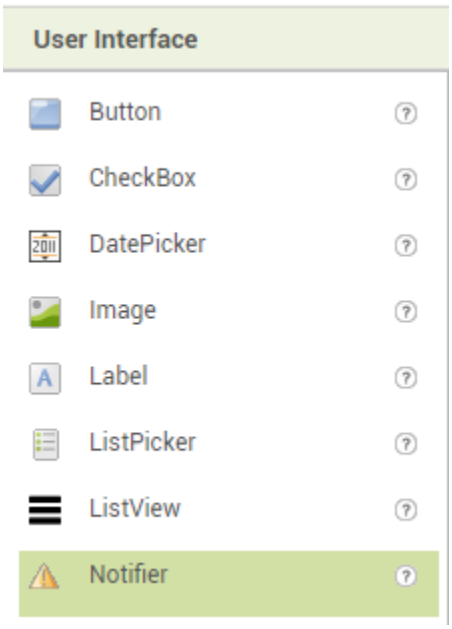
The *Text* built-in block category has a block that can be used to test if text is empty.





]

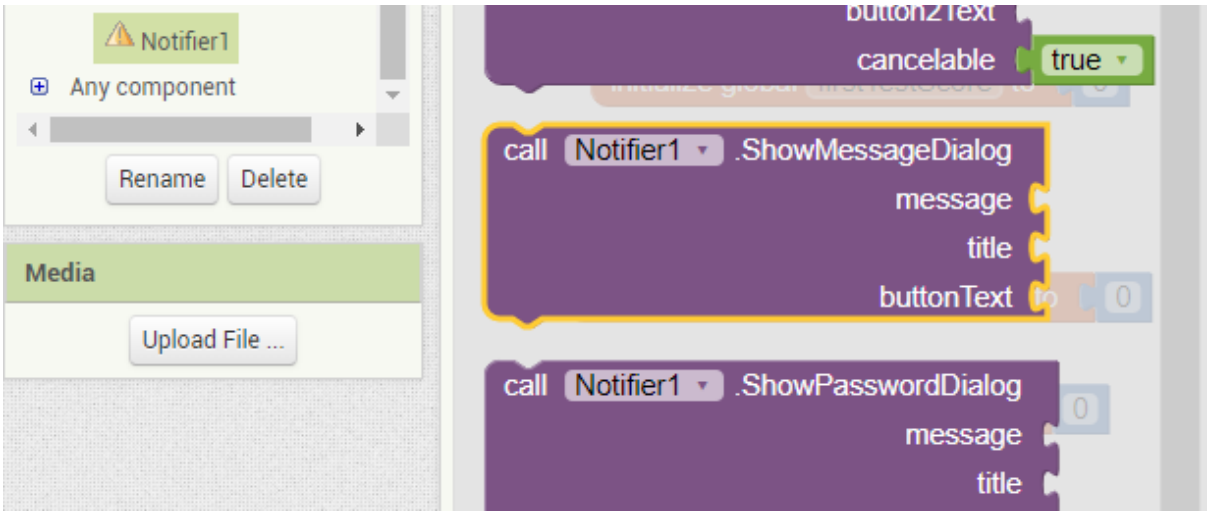
To show a popup dialog, we will use a component called “Notifier”. It can be found in the User Interface group of the Palette.

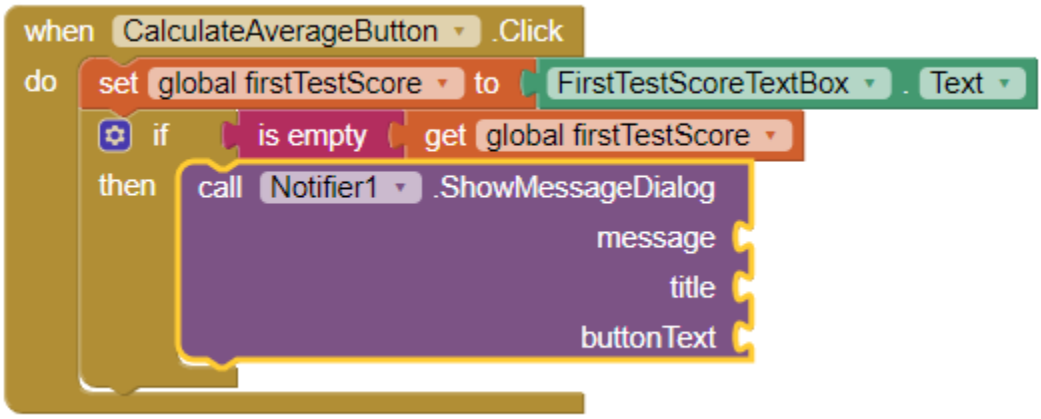


The Notifier component is a non-visible component that can be used to show different types of popups to the user. The one we are interested in for the TestAverage app is this:

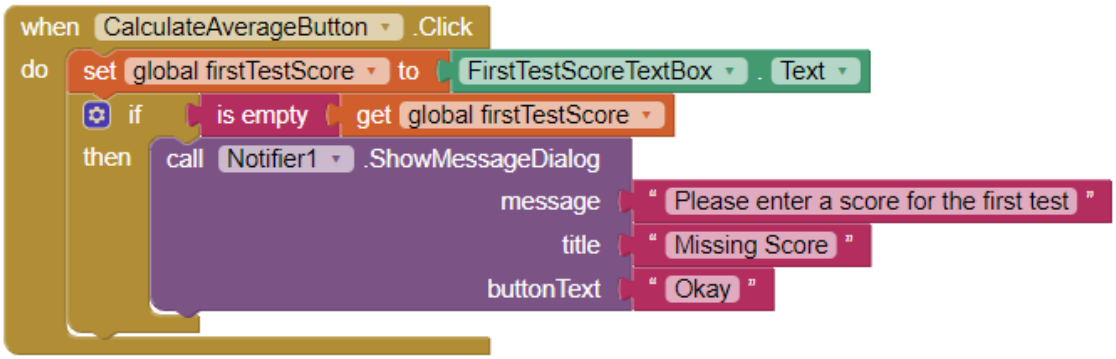
A popup dialog with a title, message and a button to dismiss the popup

To show this popup dialog, use the *showMessageDialog* command block of the Notifier component.

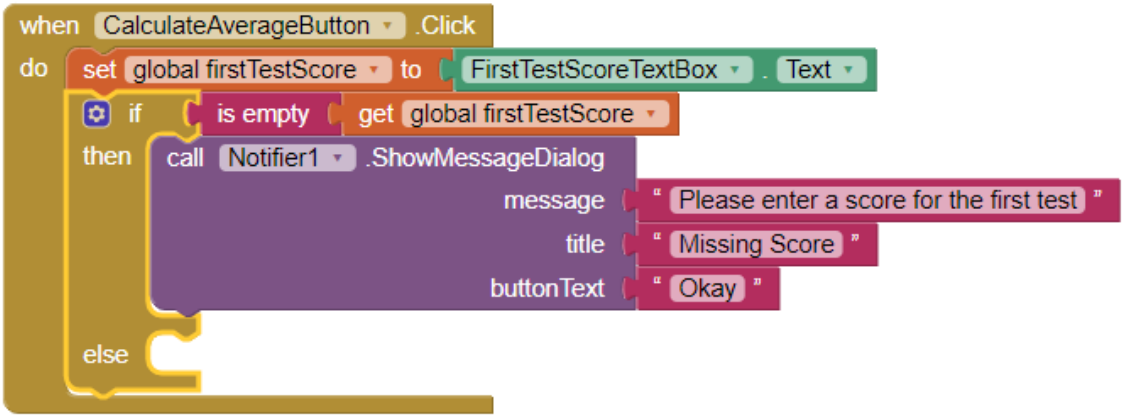




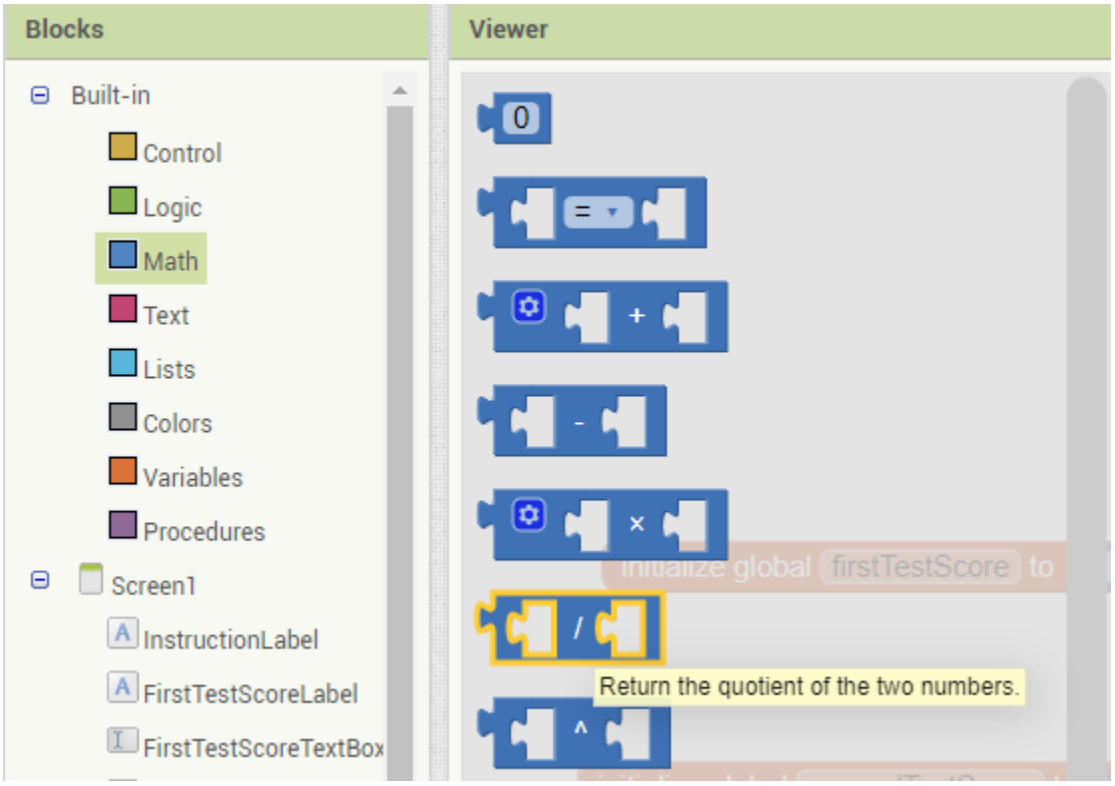
This block accepts three inputs: a message which describes the main reason for the popup, a title, and a text for the button which, when clicked, dismisses the dialog. Pass the inputs as follows:



4. Else, go on to calculate the average as follows:

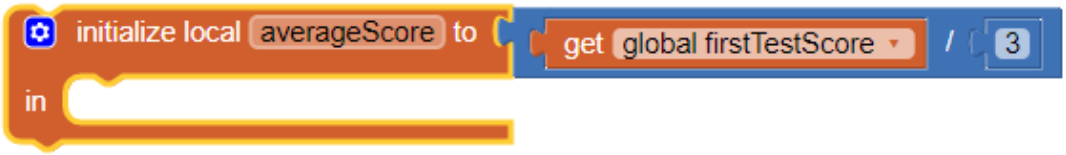


5. Divide *firstTestScore* by 3.
We will use a block from the *Math* built-in blocks category.



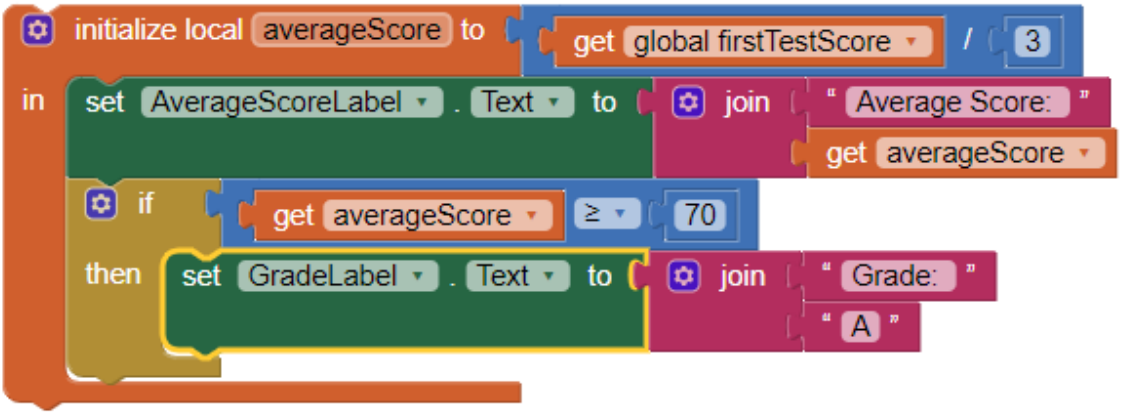
This blocks accepts two inputs, which are the two numbers that are to be divided.

6. Create a local variable *averageScore* and initialize it to the result of the division above.

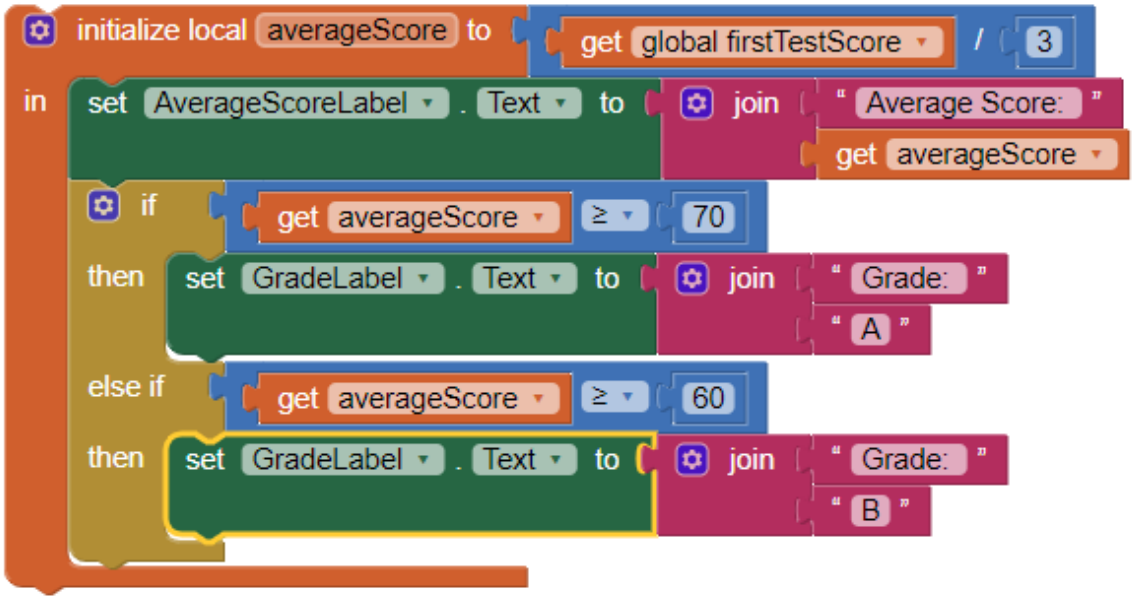


7. Inside *averageScore's* local variable block,
Set the *Text* property of *AverageScoreLabel* to "Average Score: *averageScore*".

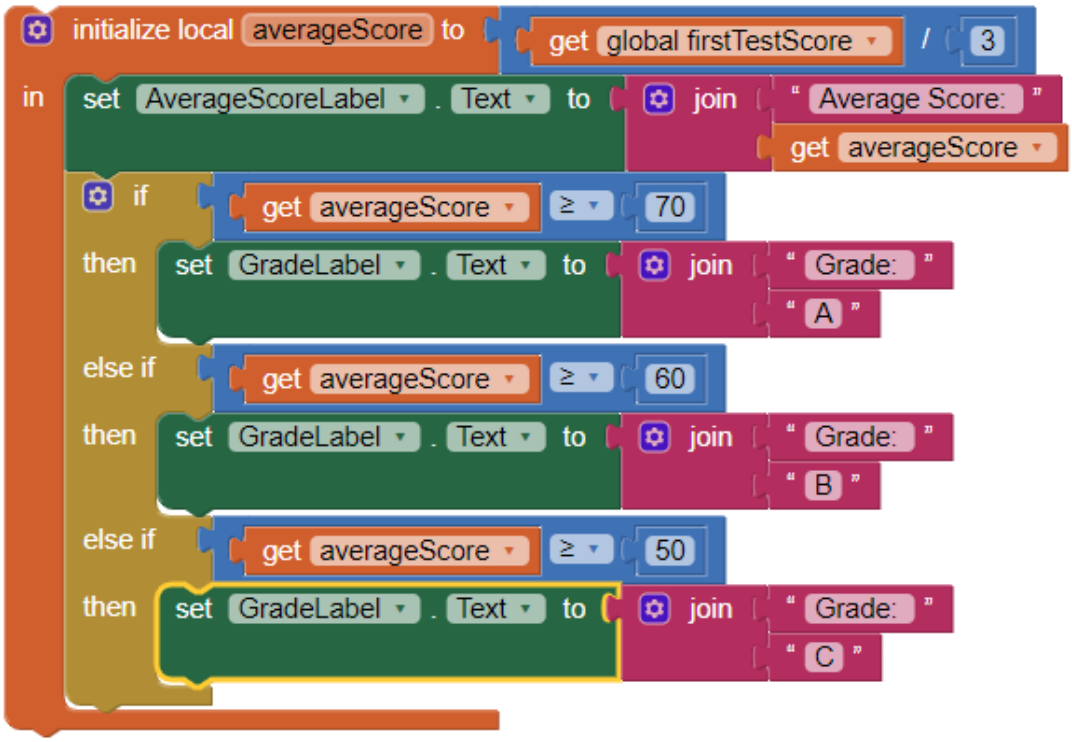
8. Test the value of *averageScore*:
If it is greater than or equal to 70, set the *Text* property of *GradeLabel* to "A".



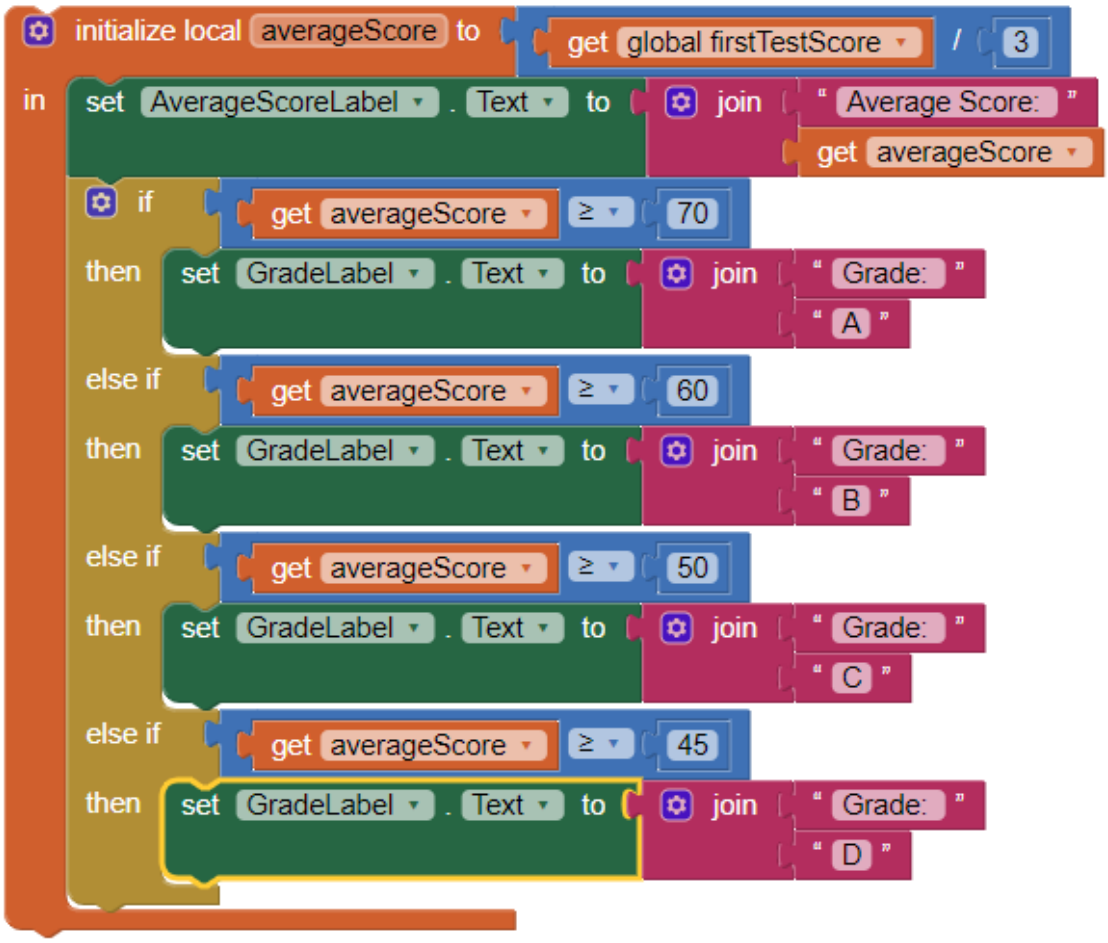
9. Else, if it is greater than or equal to 60, set the *Text* property of *GradeLabel* to "B".



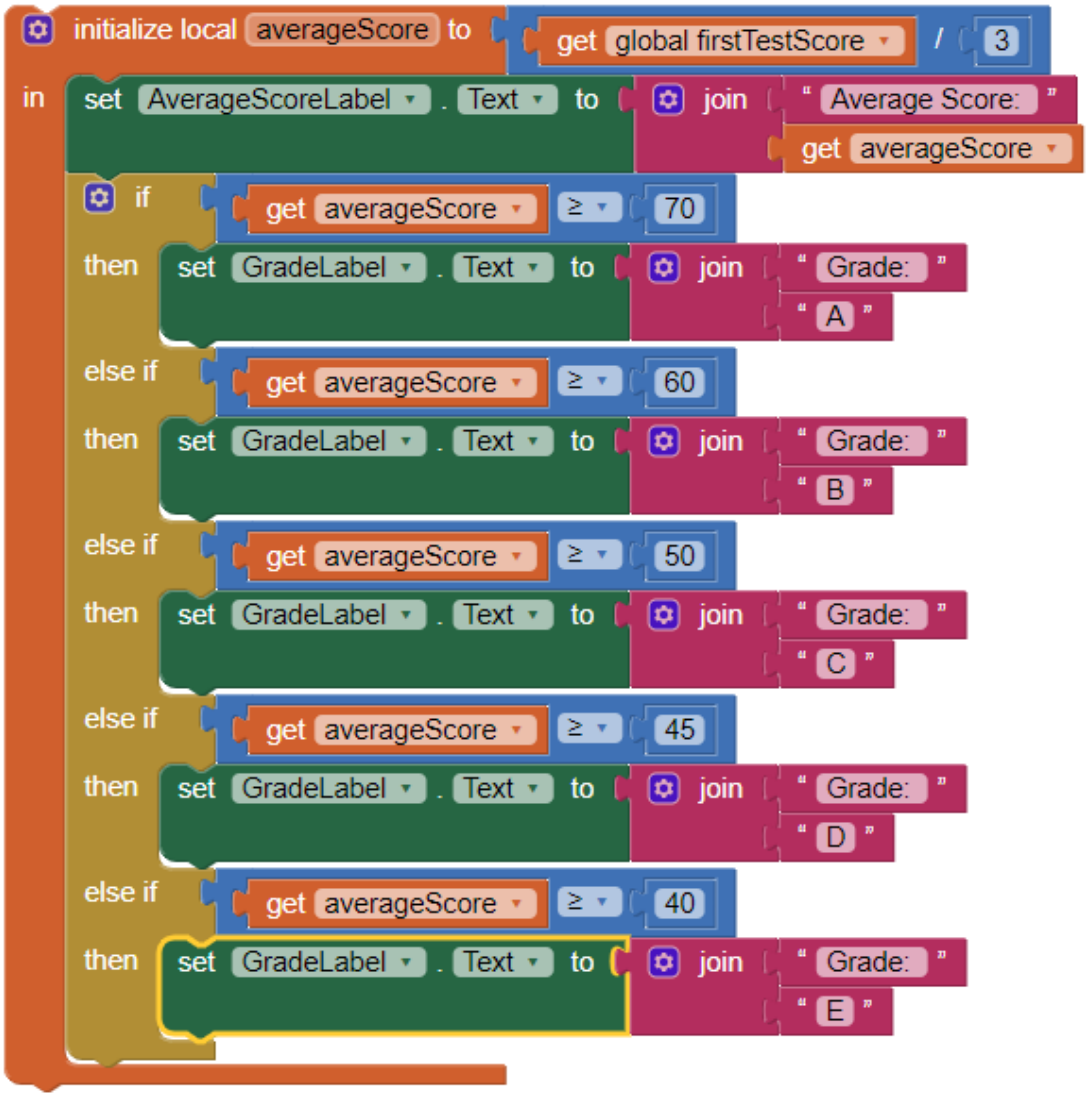
10. Else, if it is greater than or equal to 50, set the *Text* property of *GradeLabel* to "C".



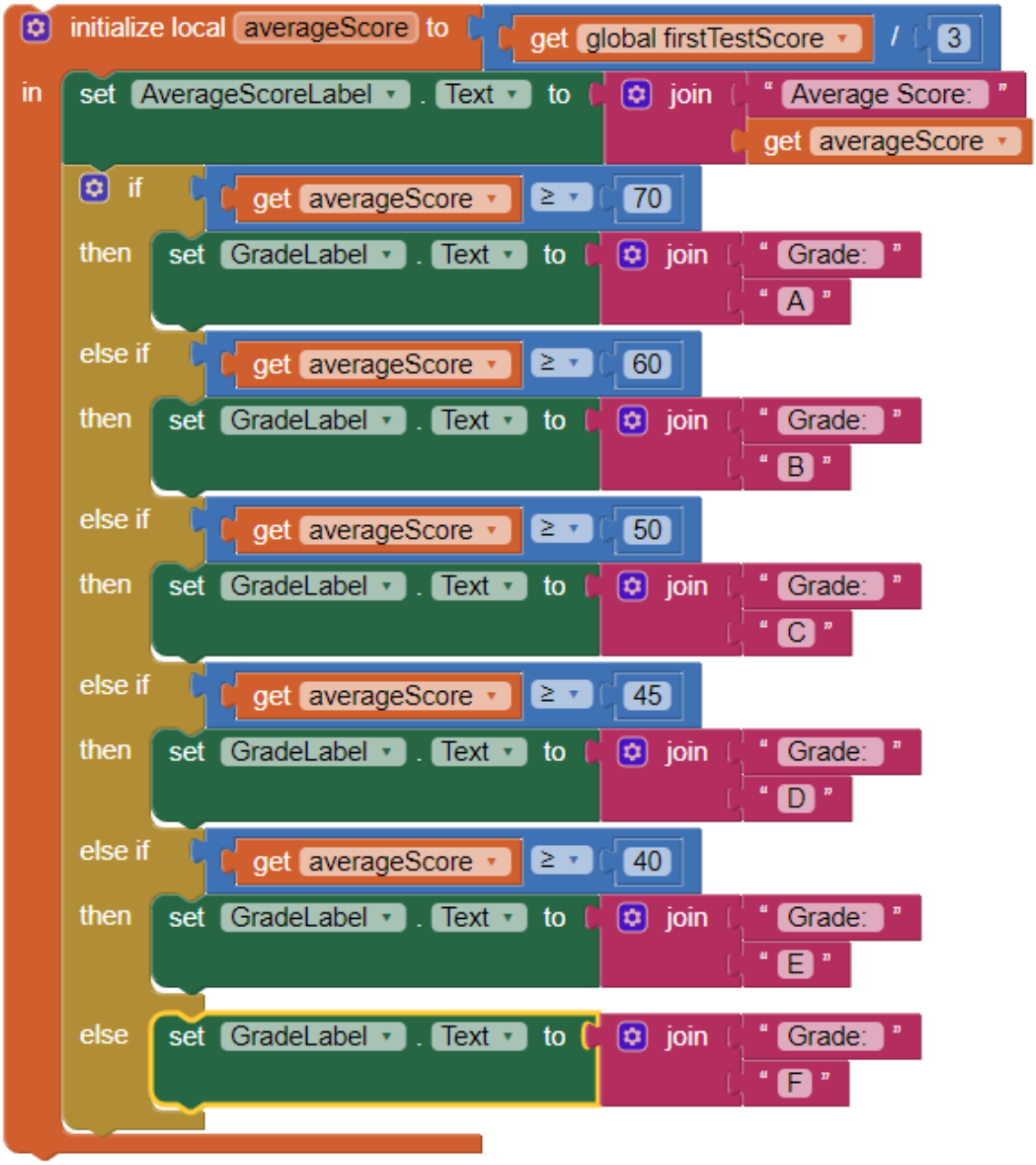
11. Else, if it is greater than or equal to 45, set the *Text* property of *GradeLabel* to "D".



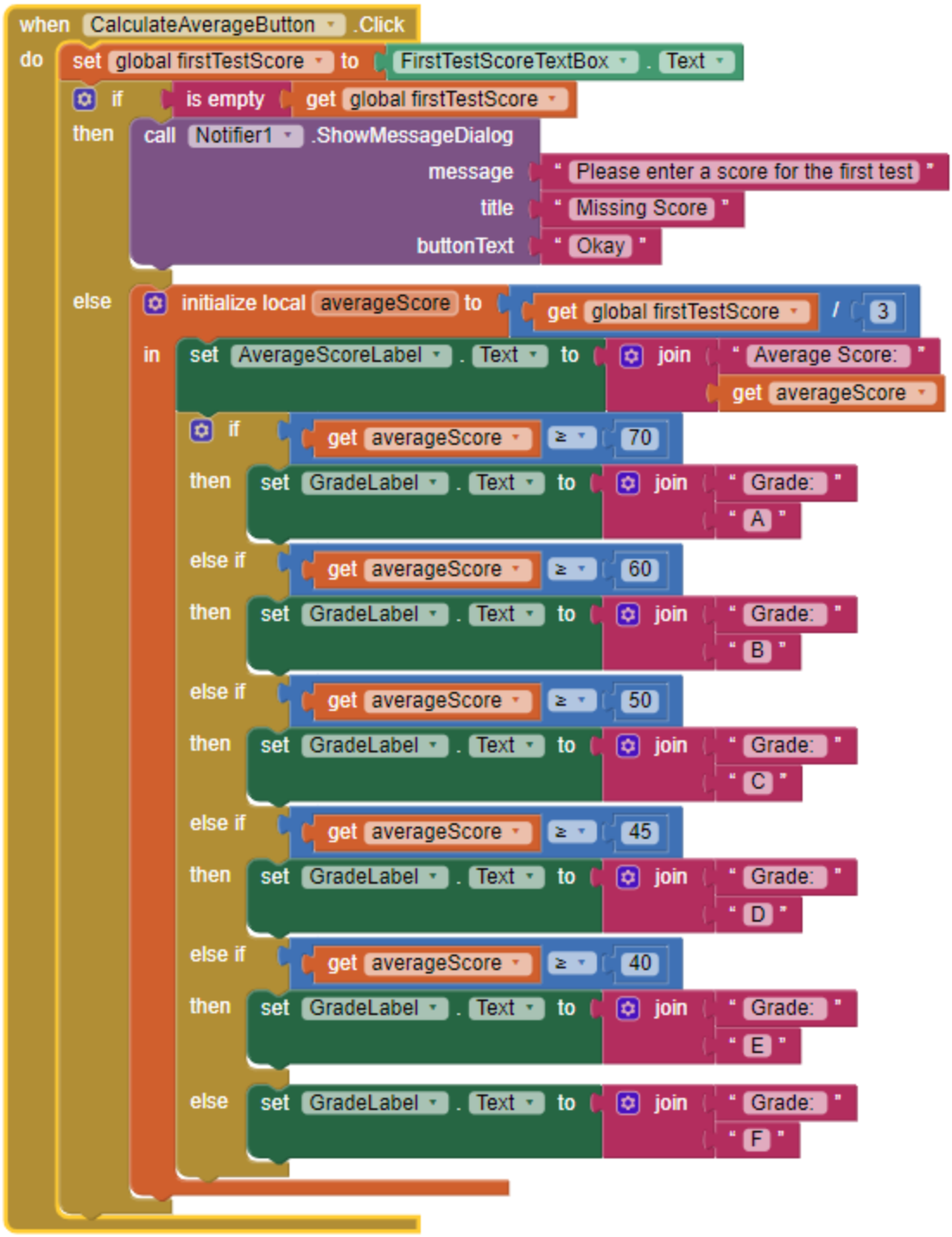
12. Else, if it is greater than or equal to 40, set the *Text* property of *GradeLabel* to "E".



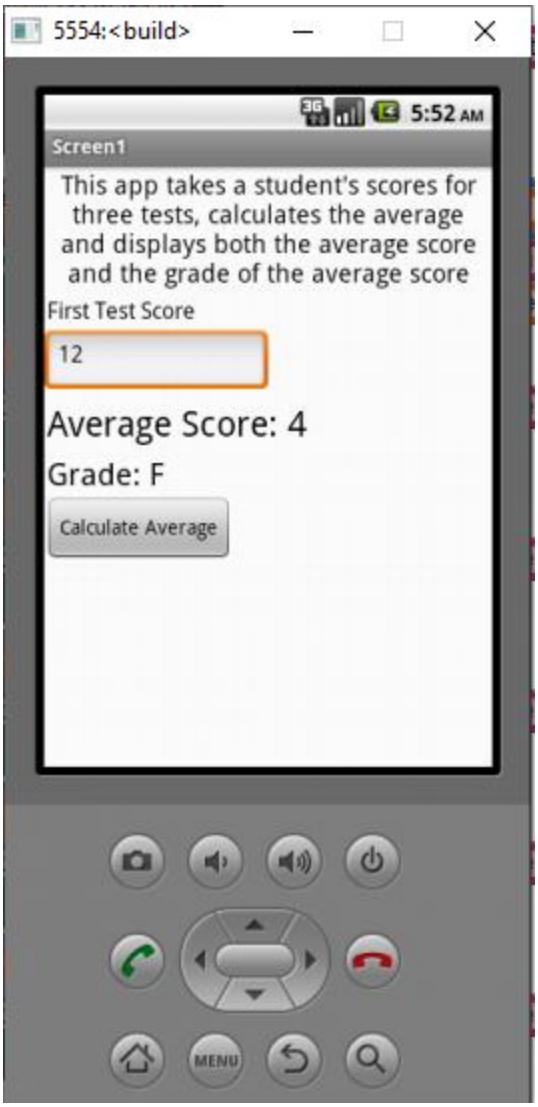
13. Else, set the *Text* property of *GradeLabel* to “F”.



Finally, we stack the local variable block in the event handler.



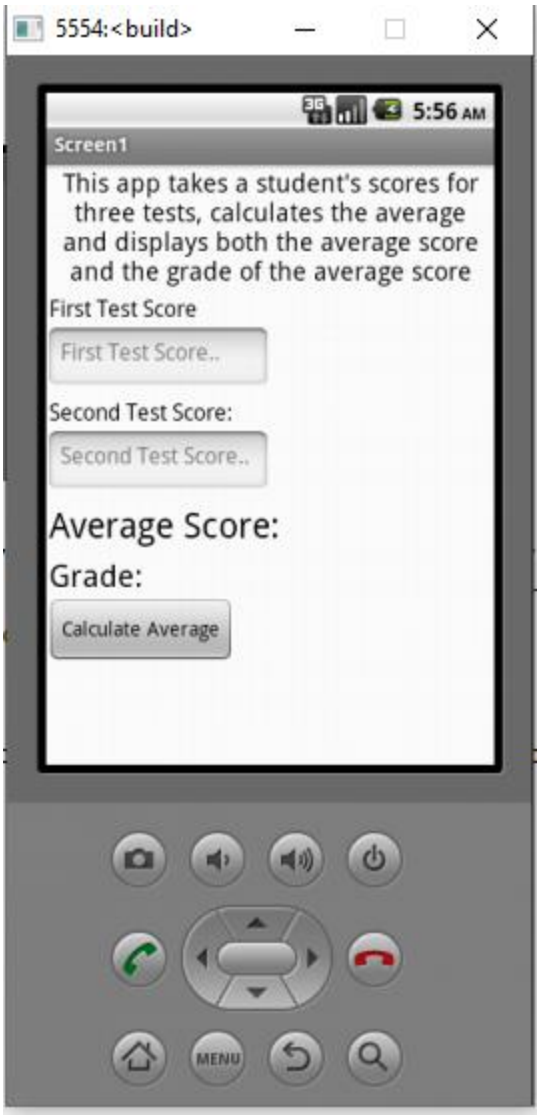
Now test the app and try to calculate the average with and without entering text in FirstScoreTextBox. Try different values and see what the average would be.



NOTE: For now, we assume that the average of a number is that number divided by 3. Make your calculations outside the app to ensure that the app is working correctly. If your average calculations do not match, check the blocks properly to see that you did not miss a step.

Extending TestAverage to Calculate the Average of Two Test Scores

We can now replicate this behaviour to calculate the average of two test scores. Modify the app UI to the following:



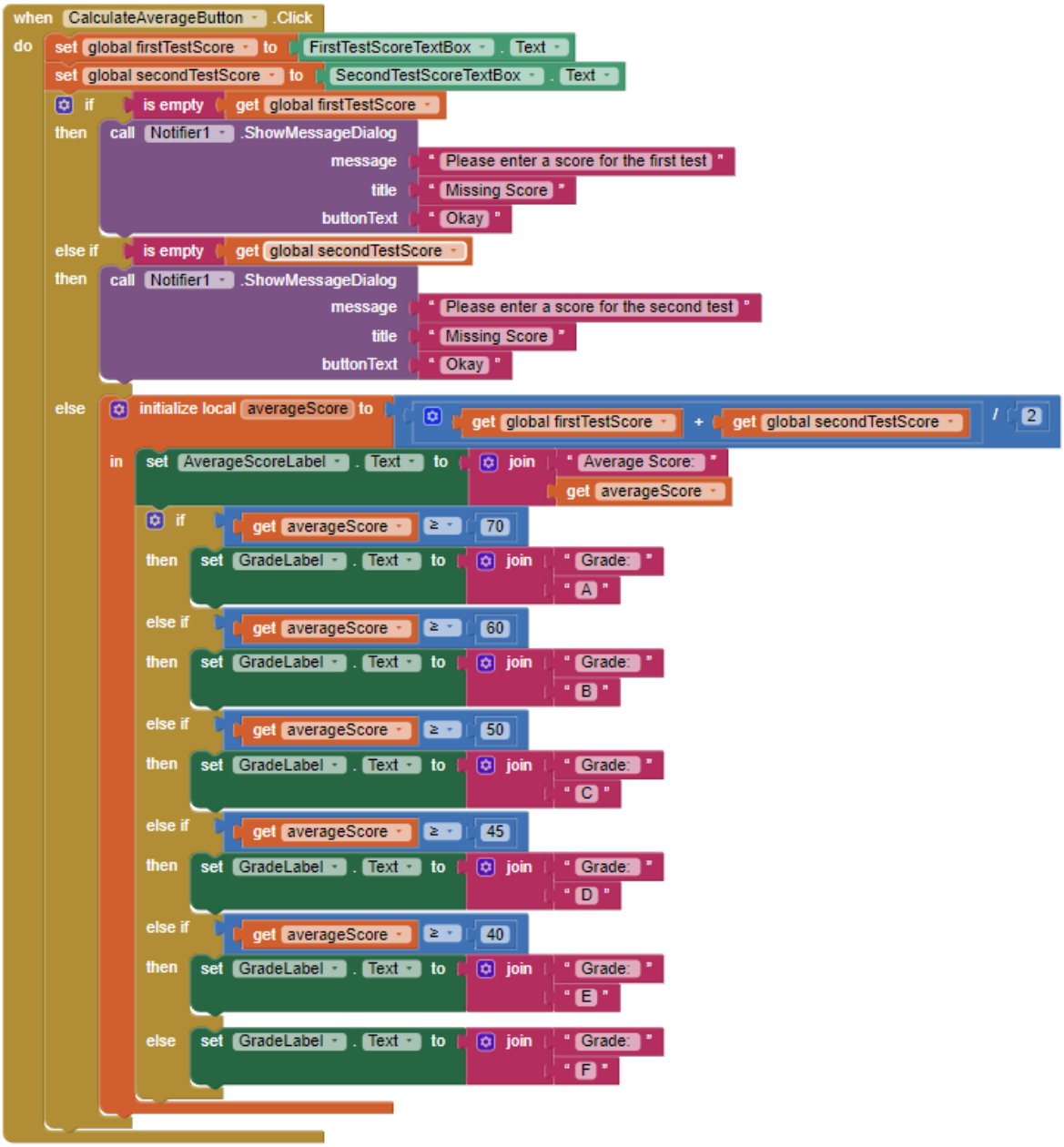
We will now calculate the average in the correct, mathematical way: add all values together and divide by the total number of values.

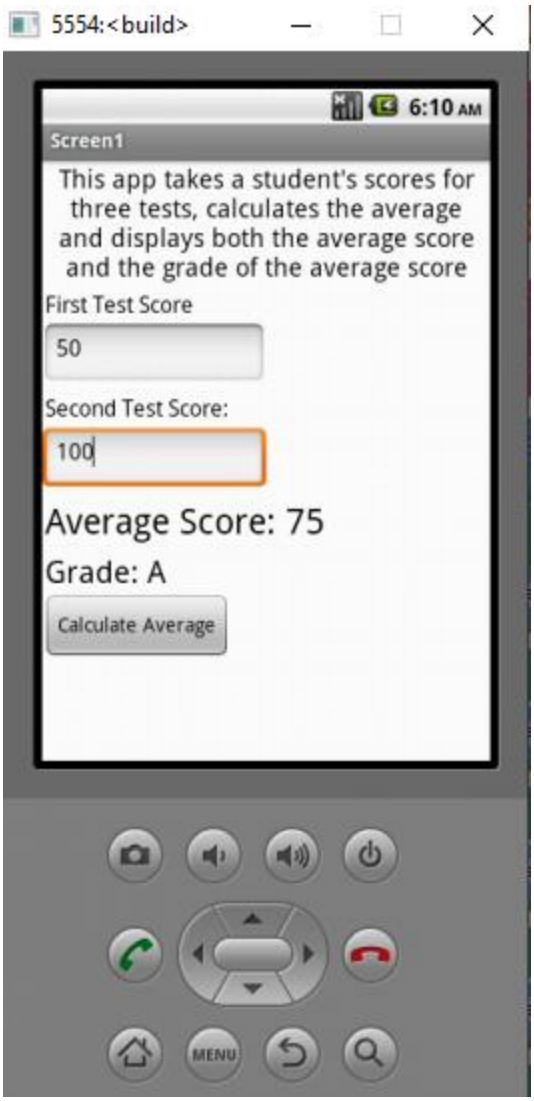
Modify the program in the Blocks Editor as follows:

- Initialize a global variable *secondTestScore* to 0.
- When *CalculateAverageButton* is clicked
    - Set the value of *firstTestScore* to the *Text* property of *FirstTestScoreTextBox*.
    - Set the value of *secondTestScore* to the *Text* property of *SecondTestScoreTextBox*.
    - Test the value of *firstTestScore*:
        - If it is empty, show a popup dialog notifying the user to enter a score for the first test.
        - Else, test the value of secondTestScore:

- If it is empty, show a popup dialog notifying the user to enter a score for the second test.
- Else, go on to calculate the average as follows:
    - Add *firstTestScore* and *secondTestScore*.
    - Divide the result of the addition above by 2.
    - Create a local variable *averageScore* and initialize it to the result of the division above.
    - Inside *averageScore's* local variable block,
        - Set the *Text* property of *AverageScoreLabel* to "Average Score: *averageScore*".
        - Test the value of *averageScore*:
            - If it is greater than or equal to 70, set the *Text* property of *GradeLabel* to "A".
            - Else, if it is greater than or equal to 60, set the *Text* property of *GradeLabel* to "B".
            - Else, if it is greater than or equal to 50, set the *Text* property of *GradeLabel* to "C".
            - Else, if it is greater than or equal to 45, set the *Text* property of *GradeLabel* to "D".
            - Else, if it is greater than or equal to 40, set the *Text* property of *GradeLabel* to "E".
            - Else, set the *Text* property of *GradeLabel* to "F".

By the time you are done with these program modifications, you should have the following blocks:

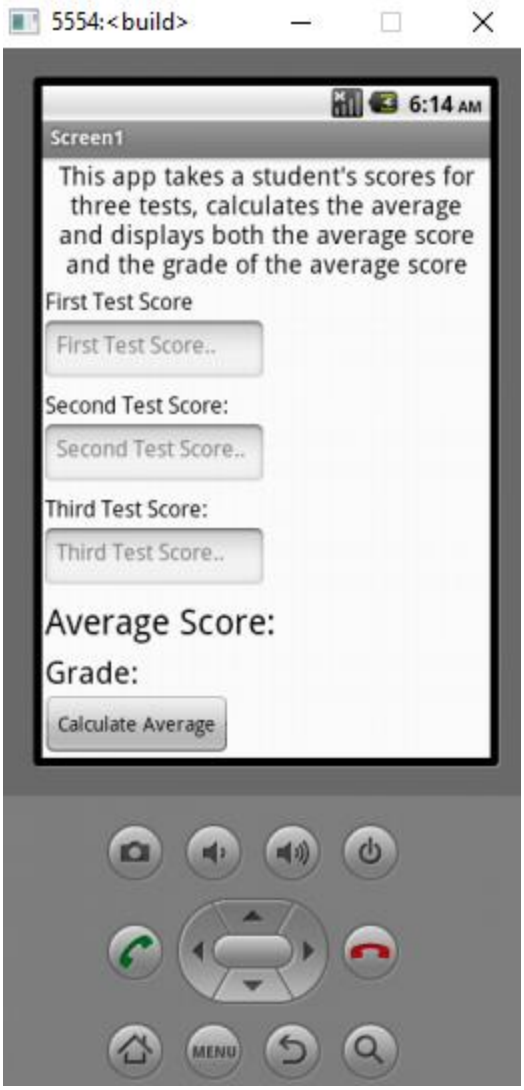

when CalculateAverageButton .Click
do set global firstTestScore to FirstTestScoreTextBox . Text
set global secondTestScore to SecondTestScoreTextBox . Text
if is empty get global firstTestScore
then call Notifier1 .ShowMessageDialog
message " Please enter a score for the first test "
title " Missing Score "
buttonText " Okay "
else if is empty get global secondTestScore
then call Notifier1 .ShowMessageDialog
message " Please enter a score for the second test "
title " Missing Score "
buttonText " Okay "
else initialize local averageScore to get global firstTestScore + get global secondTestScore / 2
in set AverageScoreLabel . Text to join " Average Score: " get averageScore
if get averageScore ≥ 70
then set GradeLabel . Text to join " Grade: " " A "
else if get averageScore ≥ 60
then set GradeLabel . Text to join " Grade: " " B "
else if get averageScore ≥ 50
then set GradeLabel . Text to join " Grade: " " C "
else if get averageScore ≥ 45
then set GradeLabel . Text to join " Grade: " " D "
else if get averageScore ≥ 40
then set GradeLabel . Text to join " Grade: " " E "
else set GradeLabel . Text to join " Grade: " " F "



Extending TestAverage to Calculate the Average of Three Test Scores

Now that the app works as expected for calculating the average of two test scores, let use extend it to calculate the average of three test scores.

Modify the app UI to this:



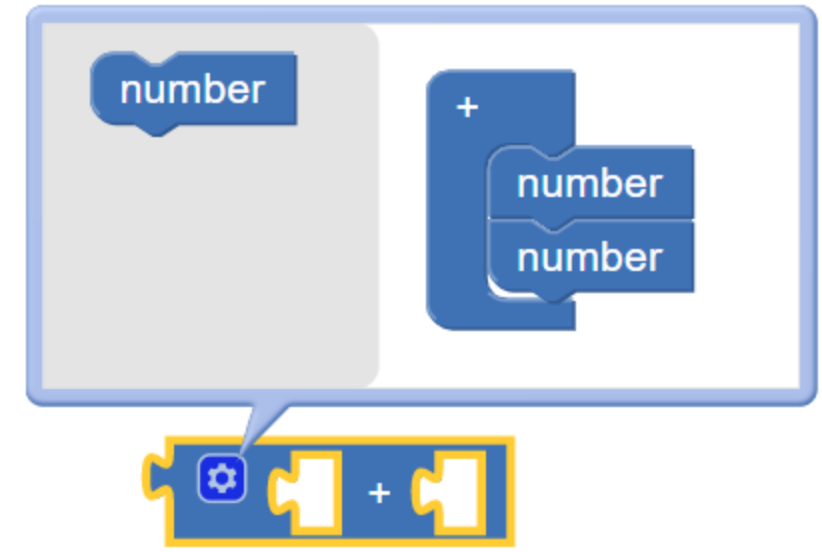
Modify the program in the Blocks Editor as follows:

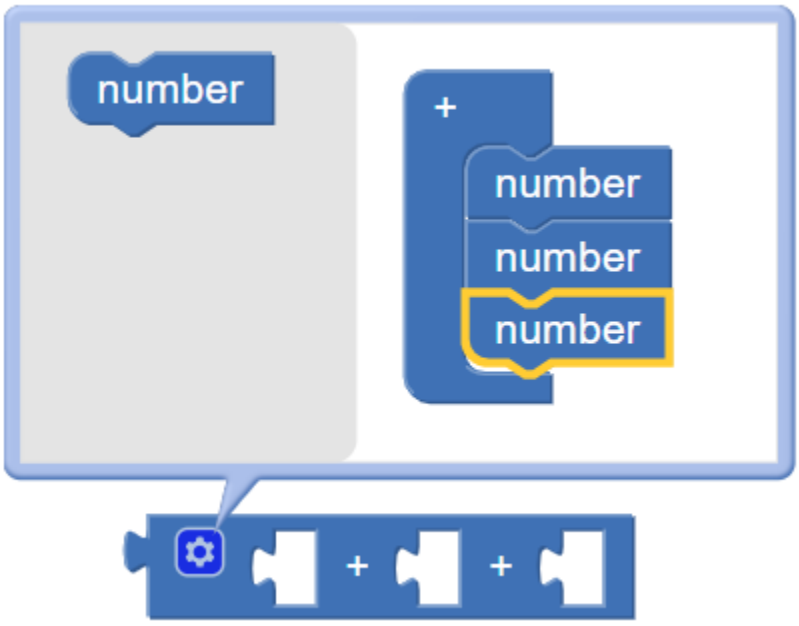
- Initialize a global variable *thirdTestScore* to 0.
- When *CalculateAverageButton* is clicked
    - Set the value of *firstTestScore* to the *Text* property of *FirstTestScoreTextBox*.
    - Set the value of *secondTestScore* to the *Text* property of *SecondTestScoreTextBox*.
    - Set the value of *secondTestScore* to the *Text* property of *SecondTestScoreTextBox*.
    - Test the value of *firstTestScore*:
        - If it is empty, show a popup dialog notifying the user to enter a score for the first test.
        - Else, test the value of *secondTestScore*:
            - If it is empty, show a popup dialog notifying the user to enter a score for the second test.
            - Else, test the value of *thirdTestScore*:

- If it is empty, show a popup dialog notifying the user to enter a score for the third test.
- Else, go on to calculate the average as follows:
    - Add *firstTestScore*, *secondTestScore* and *thirdTestScore*.
    - Divide the result of the addition above by 3.
    - Create a local variable *averageScore* and initialize it to the result of the division above.
    - Inside *averageScore's* local variable block,
        - Set the *Text* property of *AverageScoreLabel* to "Average Score: *averageScore*".
        - Test the value of *averageScore*:
            - If it is greater than or equal to 70, set the *Text* property of *GradeLabel* to "A".
            - Else, if it is greater than or equal to 60, set the *Text* property of *GradeLabel* to "B".
            - Else, if it is greater than or equal to 50, set the *Text* property of *GradeLabel* to "C".
            - Else, if it is greater than or equal to 45, set the *Text* property of *GradeLabel* to "D".
            - Else, if it is greater than or equal to 40, set the *Text* property of *GradeLabel* to "E".
            - Else, set the *Text* property of *GradeLabel* to "F".

The addition block from the *Math* built-in blocks category accepts two values for addition by default.

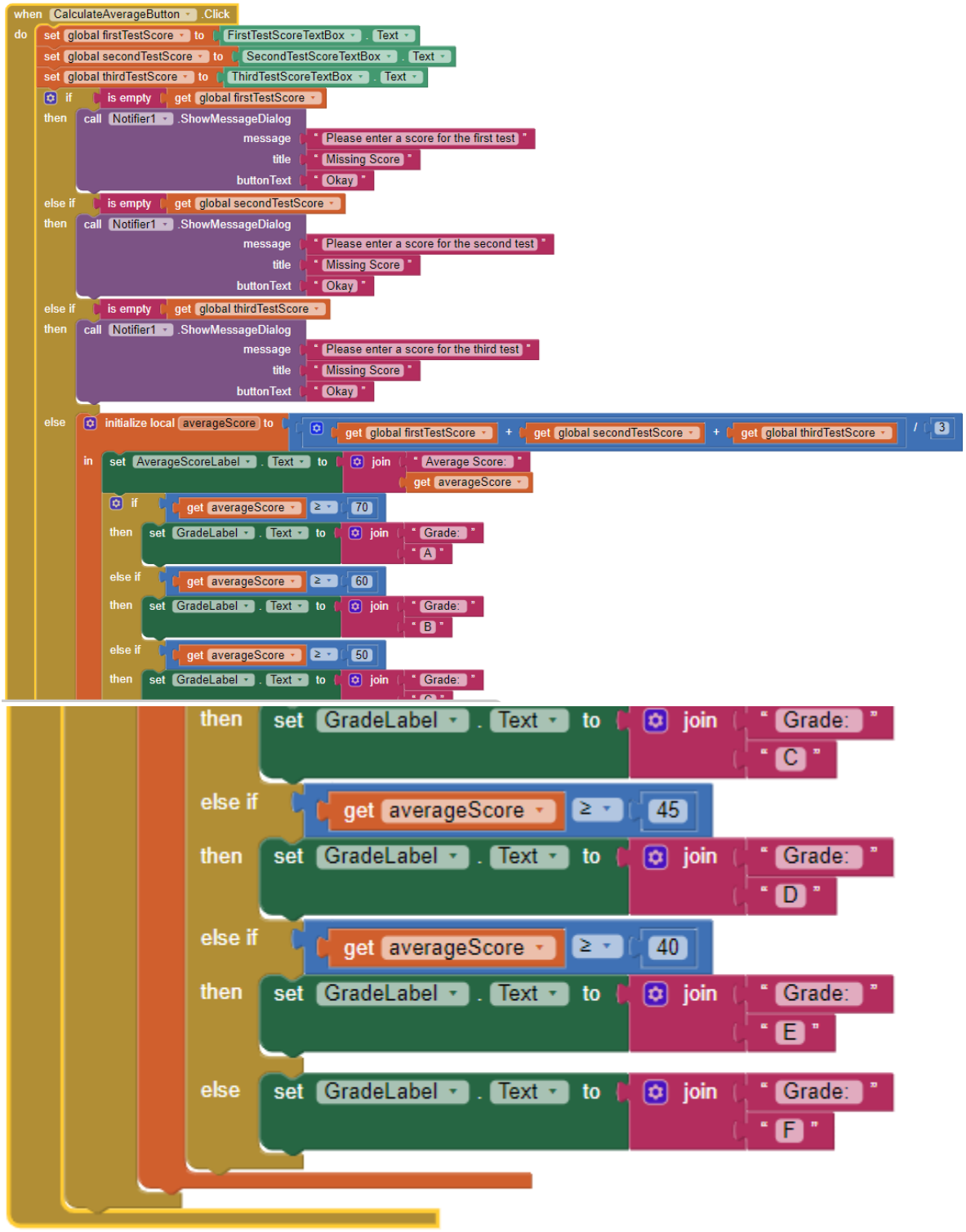
To add more than two numbers together, click on the settings icon on the block and drag as many *number* blocks from the left to the block on the right.

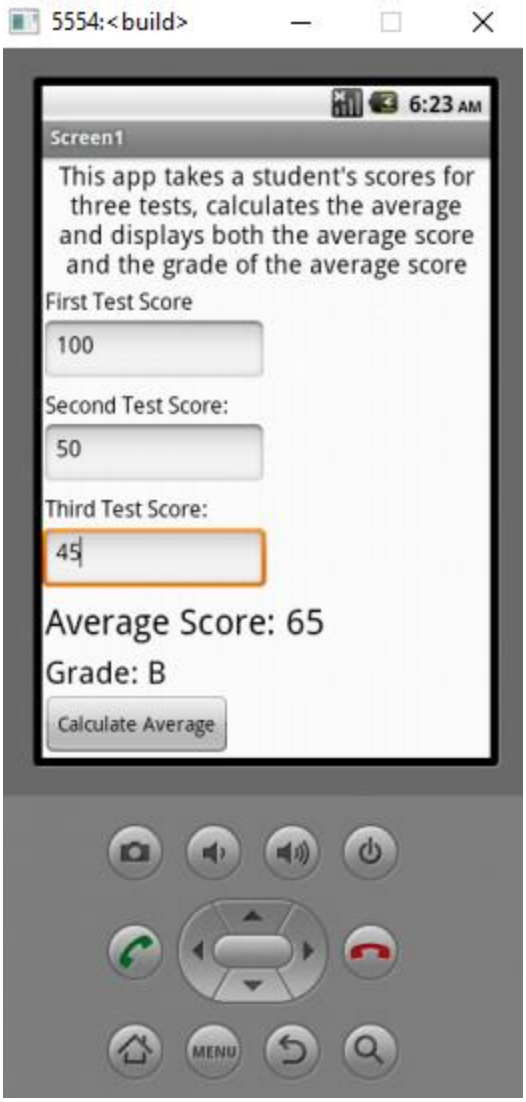




You can now perform the complex addition in the steps above.

By the time you are done with these program modifications, you should have the following blocks:


when CalculateAverageButton .Click
do set global firstTestScore to FirstTestScoreTextBox . Text
set global secondTestScore to SecondTestScoreTextBox . Text
set global thirdTestScore to ThirdTestScoreTextBox . Text
if is empty get global firstTestScore
then call Notifier1 .ShowMessageDialog
message " Please enter a score for the first test "
title " Missing Score "
buttonText " Okay "
else if is empty get global secondTestScore
then call Notifier1 .ShowMessageDialog
message " Please enter a score for the second test "
title " Missing Score "
buttonText " Okay "
else if is empty get global thirdTestScore
then call Notifier1 .ShowMessageDialog
message " Please enter a score for the third test "
title " Missing Score "
buttonText " Okay "
else initialize local averageScore to get global firstTestScore + get global secondTestScore + get global thirdTestScore / 3
in set AverageScoreLabel . Text to join " Average Score: " get averageScore
if get averageScore ≥ 70
then set GradeLabel . Text to join " Grade: " " A "
else if get averageScore ≥ 60
then set GradeLabel . Text to join " Grade: " " B "
else if get averageScore ≥ 50
then set GradeLabel . Text to join " Grade: " " C "
else if get averageScore ≥ 45
then set GradeLabel . Text to join " Grade: " " D "
else if get averageScore ≥ 40
then set GradeLabel . Text to join " Grade: " " E "
else set GradeLabel . Text to join " Grade: " " F "



Summary

In this module, we learned how to use the Notifier component can be used to create popup dialogs in an app. We applied our knowledge of variables, control blocks, boolean operations and the Notifier component to build the TestAverage app, an application that can be used to calculate and display the average score and grade of any number of a student's test or exam scores.