
MUO
HOW TO CREATE
AN ANDROID APP
EVERYTHING YOU NEED TO KNOW
BY AARON PETERS

How to Create an Android App: Everything You Need to Know

Written by Aaron Peters

Published August 2017.

Read the original article here: **http://www.makeuseof.com/tag/make-your-own-android-app-your-unofficial-intro-to-mit-app-inventor/**

Table of contents

Welcome to MakeUseOf's guide to creating your own Android app. In this guide we'll take a look at why you'd want to create an Android application of your own, some options you have for building it, and how to make it available to others.

Introduction to Android Development

There are two primary ways to develop an Android app. The first is to write it from scratch, most likely in Java. But this, of course, assumes you already **know** Java or have the patience to learn itbefore diving in. But what if you're itching to get started right away?

The other option is one of the point-and-click app builders on the market. Many of these target enterprise users (and come with an enterprise price tag). But MIT offers its "App Inventor," an online tool that allows you to build your app visually. You can accomplish some neat things with App Inventor, which will keep you busy until you can dig into Java and access all the powerful features of the Android platform.

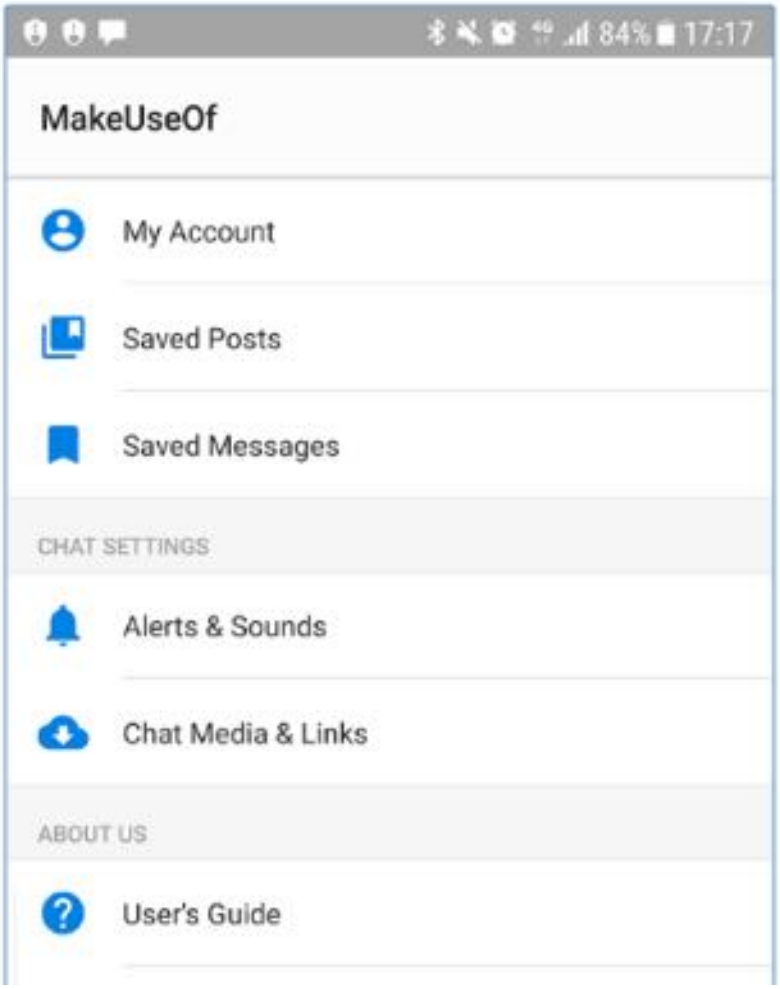
In the sections below, we'll build a prototype version of a simple "scratchpad" application, which will store the text you type into it. We'll do this first in App Inventor and preview the results in an Android emulator. Then we'll extend this application with the ability to select from among multiple files, making it more of a "notepad." For this type of improvement, we'll need to dive into Java and Android Studio.

Ready? Let's get to it.

Why Develop for Android?

There are any number of reasons you'd want to create your own Android app, including:

- **Necessity**: It's the mother of invention, after all. Maybe after looking in the Play Store for your dream app, you realize that it's something you'll need to build yourself because no one else has yet.
- **Community**: Developing something useful and making it available for free (particularly as open source) is an excellent way to participate in the Android and/or FOSS community. Without open source contributions, there would be no Linux, and without Linux there would be no Android (or at least no Android as we know it). So consider giving back!
- **Learning**: There's are few better ways to gain an understanding of a platform than to develop for it. It could be for school or your own curiosity. And hey, if you can make a couple bucks off it in the end, all the better.
- **Monetization**: On the other hand, maybe you're going at this to make money from the start. While Android was once considered the "low-rent" district of app revenues, this has slowly been turning around. Business Insider reported in March that Android revenues are expected to overtake iOS for the first time in 2017.
- **Add-on**: Developers often create apps in general as a way to promote, access, or otherwise complement an existing product or service — like console companion apps and MakeUseOf's own app.



Whatever your reason, app development will challenge your design, technical, and logical skills. And the result of this exercise (a working and useful application for Android) is a great accomplishment that can serve as a portfolio piece.

There are many avenues to creating your app, including different toolkits, **programming languages**, and **publishing outlets**. At a high level, these break down into the following two categories.

Point-and-click Apps

If you are a complete newbie to development, there are environments that let you build an Android app the same way you'd create a Powerpoint presentation. You can select controls such as buttons or text boxes, drop them onto a screen (as shown in the image below), and provide some parameters on how they should behave. All without writing any code.

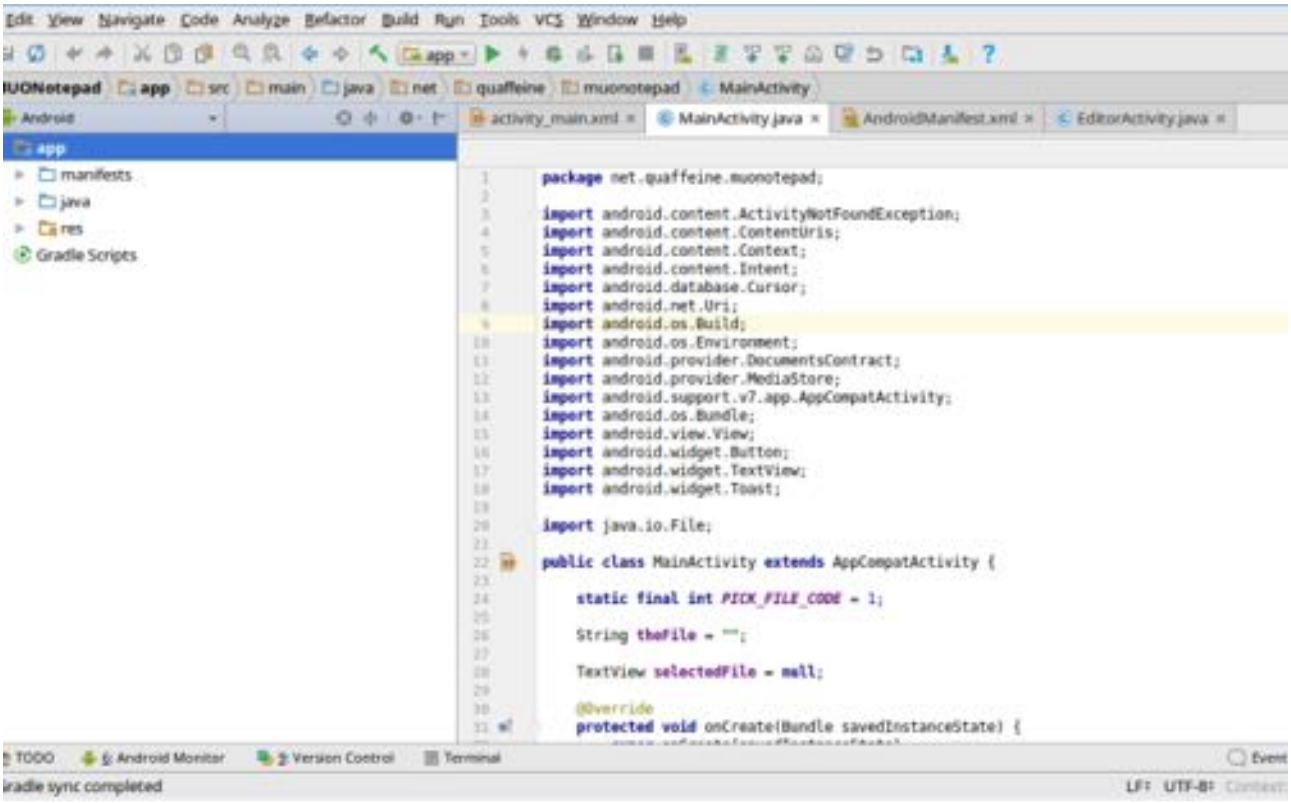
These types of applications have the advantage of a shallow learning curve. You can typically jump right in and at least begin laying out your screen. They also take a lot of complexity out of the application, as they're designed to handle technical details (like object types or error handling) behind the scenes. On the other hand, that simplicity means you're at the mercy of the tool's creator as to what features are supported. In addition, a lot of these tools are targeted at large companies and can be expensive.

An exception is **MIT's App Inventor** web application, which is functional and free. After signing in with a Google account, you can click together an app in a couple minutes, and preview it either on your phone or via an Android emulator.

Write from Scratch

The other option is to write your application from scratch. This is probably different from what you're imagining — it's **not like the movies depict it**.

It's typing code one line at a time into source files, then **compiling them** into an executable application. While it may sound boring, in reality, much more of your time in programming is spent in **design**, or thinking through how things should work. Ask most developers, and they'll say they spend only 10-15% of their time on code entry. So you'll spend most of your time daydreaming (productively) about what your app should do.



You can **code Android applications in a couple different ways**. The "standard" way is to write apps in Java, consistently one of the most popular languages in the world, although Google is adding Kotlin as another option. For performance-intensive apps such as games, you have the option of writing in a "native" language such as C++. These apps run directly on the hardware of your Android device, as opposed to "regular" Java-based apps that run on the Dalvik **Virtual Machine**. Finally, there are ways of "wrapping up" web applications (using toolkits such as Microsoft's Xamarin or **Facebook's Native React**) for distribution as mobile apps that look "native."

While **integrated development environments (IDEs)** do handle some of the routine elements of programming, understand that the learning curve for this method is steep. Whatever language you choose, you'll need to be versed in its basics. Investing this time up front is a drawback of this method, in the sense that you won't be able to get into the development of your app right away. But it's an advantage in the long run, as the skills you learn can be applied elsewhere. **Learn Java**, and you can develop for desktop and server-side applications (including web-based ones) in addition to Android apps.

Which Option is Best For Your Project?

So which avenue is the "best?" This is too subjective to answer for everyone, but we can generalize it as follows. If you're curious but just "playing around," stick with the point-and-click app creators. They'll help you scratch that creative itch without requiring any "coursework." But if the idea of that coursework doesn't frighten you, consider taking the longer path and learning a programming language. The investment will pay off in many other ways down the line.

In addition, consider using both! Point-and-click builders are an excellent way to quickly put together a prototype or "proof of concept." Use them to work through some of the details (like layout and screen flow), as they are **much** quicker to shuffle around in a mouse-driven environment. Then re-implement them in Java if needed to take advantage of its flexibility.

We'll take precisely that approach in this guide. We will:

1. **Prototype** our application, a “scratchpad” that will store some text in a file for you, using MIT’s App Inventor.
2. **Re-implement** this in Java (with a little help from Google’s Android Studio IDE), then go on to **extend** the app to allow you to select from among multiple files, making it more of a “notepad.”

Alright, enough talking. In the next section, we’ll get ready to code.

Getting Ready to Create Your App

Don’t dive right in just yet — first you need some knowledge and some software.

Knowledge You’ll Need

Before we start installing some software, there’s some knowledge you should have before you start. First and foremost is, **“What’s it supposed to do?”** Waiting until you have a clear concept for your app before starting development may seem like a given — but you’d be surprised. So take some time to work through this concept, even jotting some notes on behavior and sketching some screens. Have a relatively complete picture of your app first.

Next, look into **what’s possible.** For example, imagine the ideal picture of your app is something that lets you video-log your entire life for posterity. You **can** create an app that will capture video. You **can’t** create one that will store every moment of your life on your device (insufficient storage). However, you **can** try to offload some of this storage to the cloud, although it will take time to develop, and that comes with its own limitations (what happens when you have no network access?). This is where you’ll examine some of the technical details and can inform decisions like whether you’ll code from scratch or not.

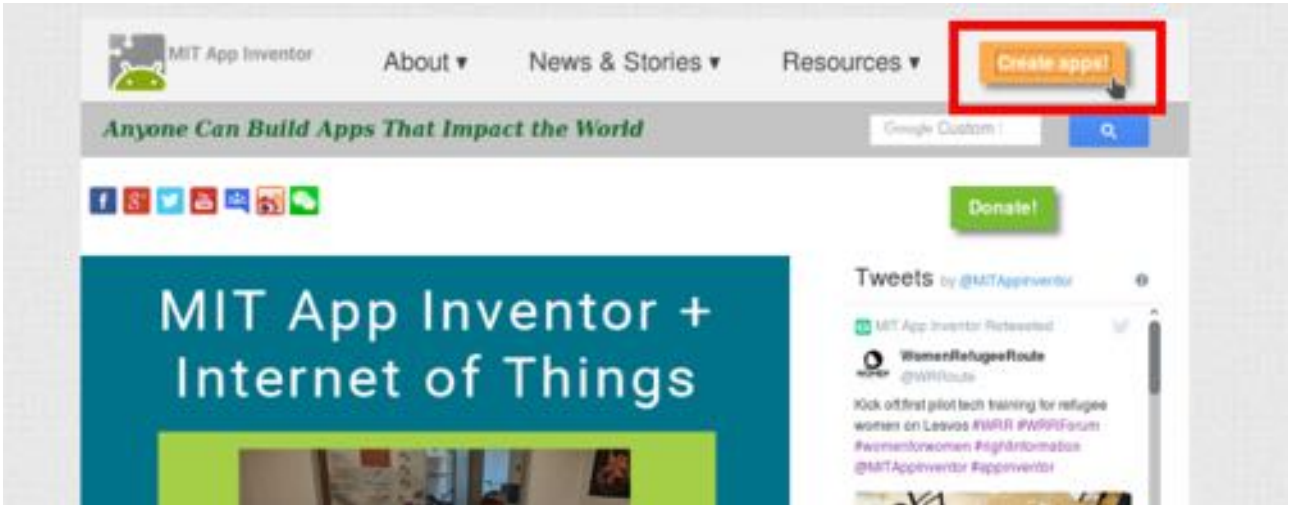
Lastly, it’s worth knowing **what’s out there** already. If you’re just looking to learn or contribute to the community, is there an existing open source project like yours? Could you fork that project as a starting point? Or even better, develop your enhancement and contribute it? If you’re looking to make money, what’s your competition like? If you write a simple alarm clock app and expect to make a million dollars off it, you had better bring something special to the table.

As discussed, we’ll be building a simple scratchpad, which collects and holds some text you put into it. And in doing so, we’ll be breaking the rules above, since there are many Android note-taking apps out there already, both open and closed source. But let’s pretend this will become a much more complex app later. You’ve got to start somewhere.

Now we’ll get some of the software you’ll need.

Preparing to Develop with App Inventor

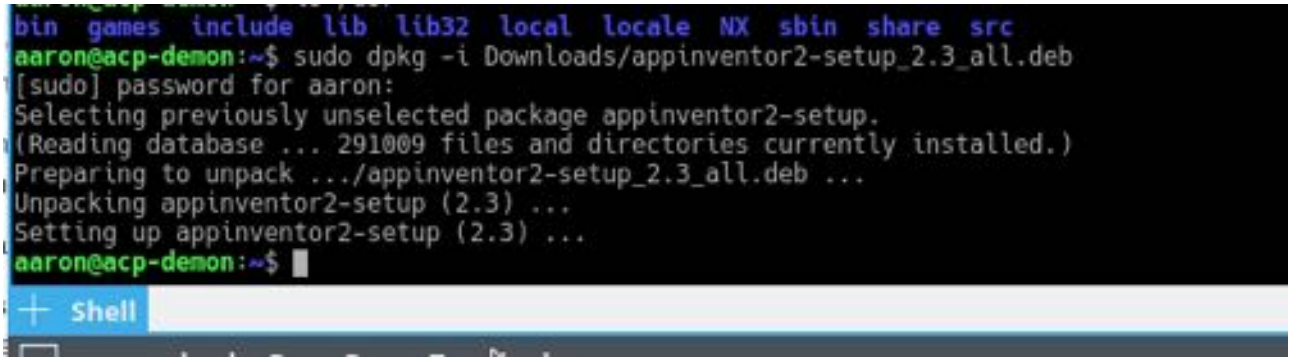
You don’t need to install anything to use the App Inventor tool. It’s a web application, and you access it entirely through the browser. When you visit the site, you’ll see a button in the upper-right corner to **Create apps!** If you’re not currently logged into a Google account, clicking on this will direct you to a log-in page.



Otherwise you should go directly to the App Inventor's **My projects** page.

At this point, consider where you want to test your app. If you're adventurous, you can test it out on your phone or tablet by installing **the Companion app from the Play Store**. Then you're all set for now — you'll need a running project to actually see anything on your device, but we'll get to that later.

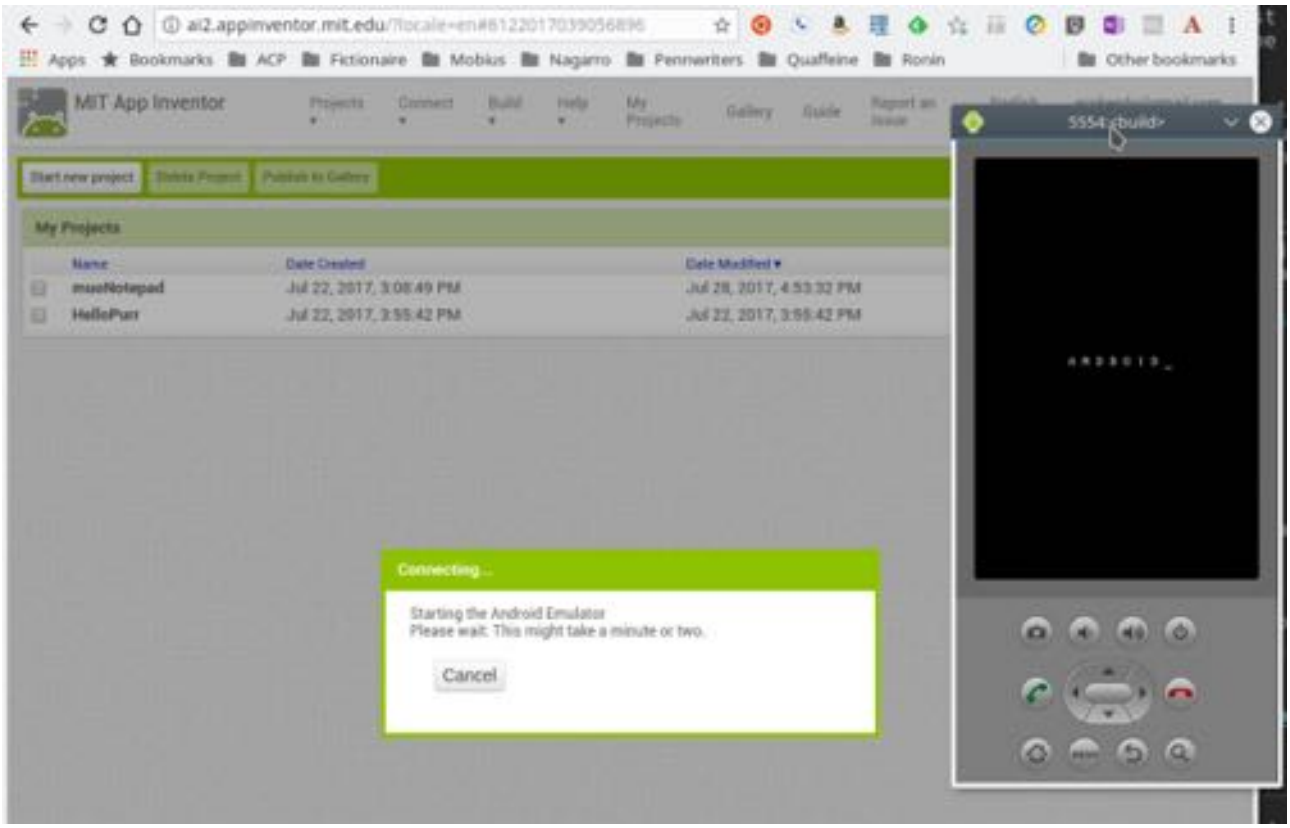
Alternatively, you can use the emulator to test your app on your computer. Download and install the emulator for your operating system from **this page**. The below image shows the app installing on Linux, but the appropriate version should install without issue on Windows or Mac as well.



You can start the emulator by running the "aiStarter" command. This starts a background process that connects your (local) emulator to the (cloud-based) App Inventor. Windows systems will provide a shortcut for it, while it will start automatically for Mac users on login. Linux users will need to run the following in a terminal:

```
/usr/google/appinventor/commands-for-appinventor/aiStarter &
```

Once it's running, you can test the connection by clicking on the **Emulator** item in the **Connect** menu. If you see the emulator spin up (as shown in the image below), you're good to go.



Installing Android Studio

If you're planning to develop some simple programs, App Inventor may be all you'll ever need. But after playing around with it for a while, you may hit a wall, or you might know you'll be using some features that App Inventor doesn't support (like in-app billing). For this, you'll need to have Android Studio installed.

Now the official development environment as sanctioned by Google, Android Studio is a version of the IntelliJ IDEA Java IDE from JetBrains. You can download a copy for your operating system from Google's Android Developer page here. Windows and Mac users can launch the installer using an EXE file or DMG image, respectively.

Linux users can use the ZIP file, unpack it wherever you like, and run Android Studio from there (Windows/Mac users can also do this). Otherwise, you can use **Ubuntu Make** to download and install the package for you. If you're on the most recent LTS version (16.04 as of this writing), you'll need to add the Ubuntu Make PPA to your system to get access to Android Studio:

```
sudo add-apt-repository ppa:ubuntu-desktop/ubuntu-make
```

Then update your system with the following.

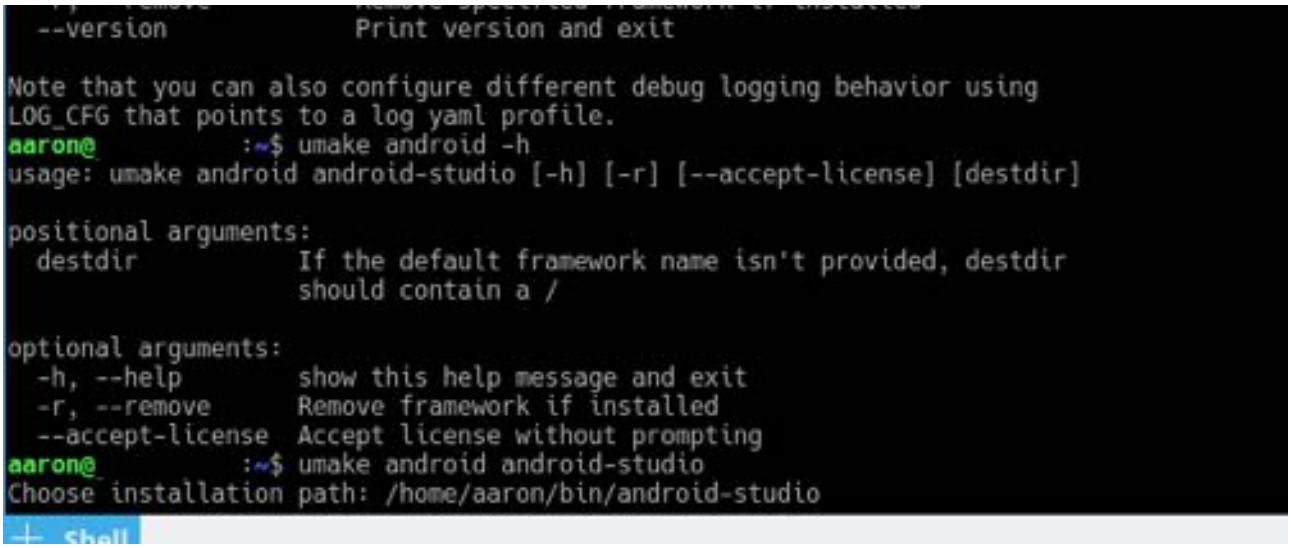
```
sudo apt update
```

Lastly, install Ubuntu Make with this command:

```
sudo apt install umake
```

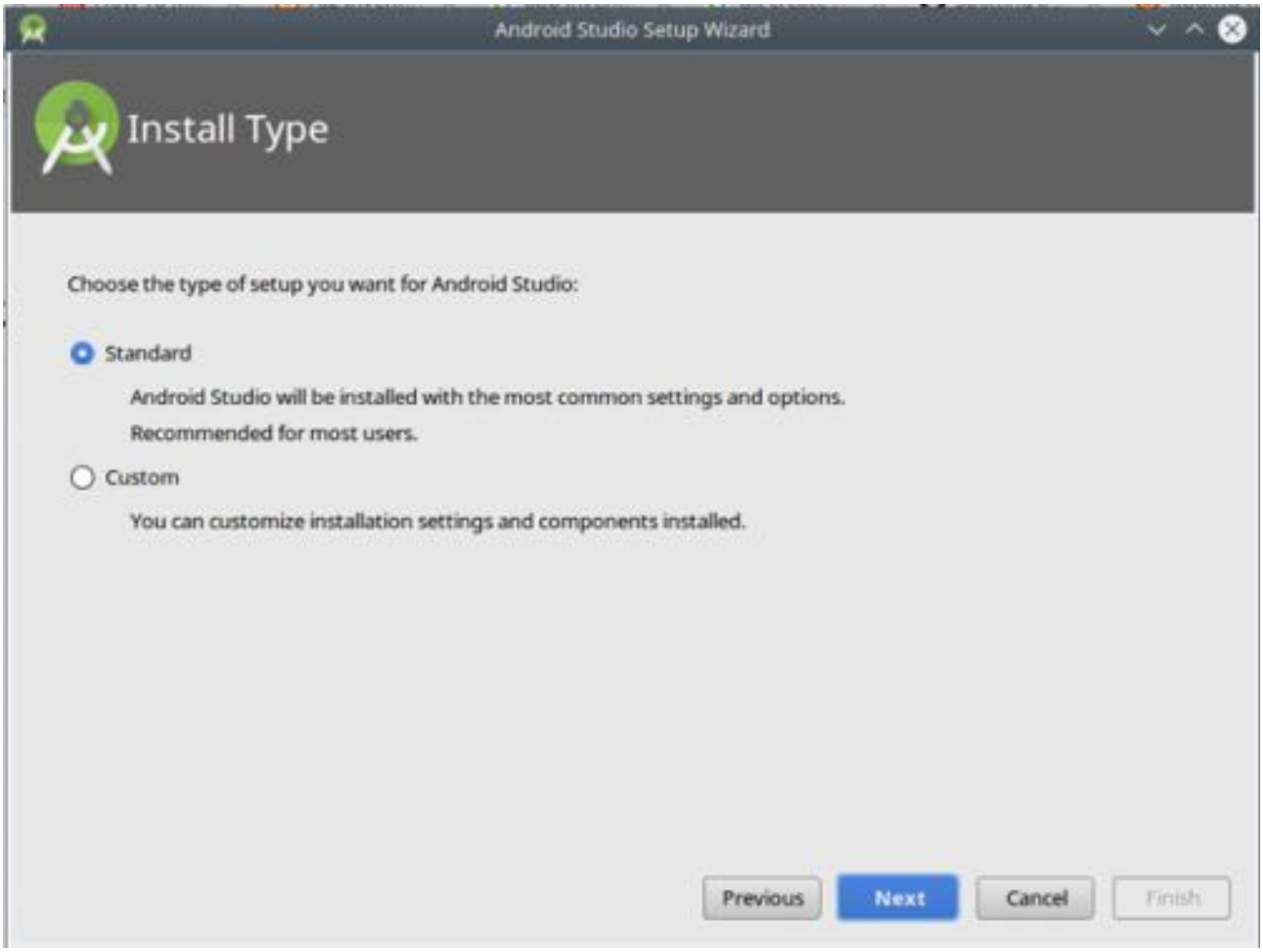
Once installed, you can direct Ubuntu Make to install Android Studio for you with the following command:

```
umake android android-studio
```

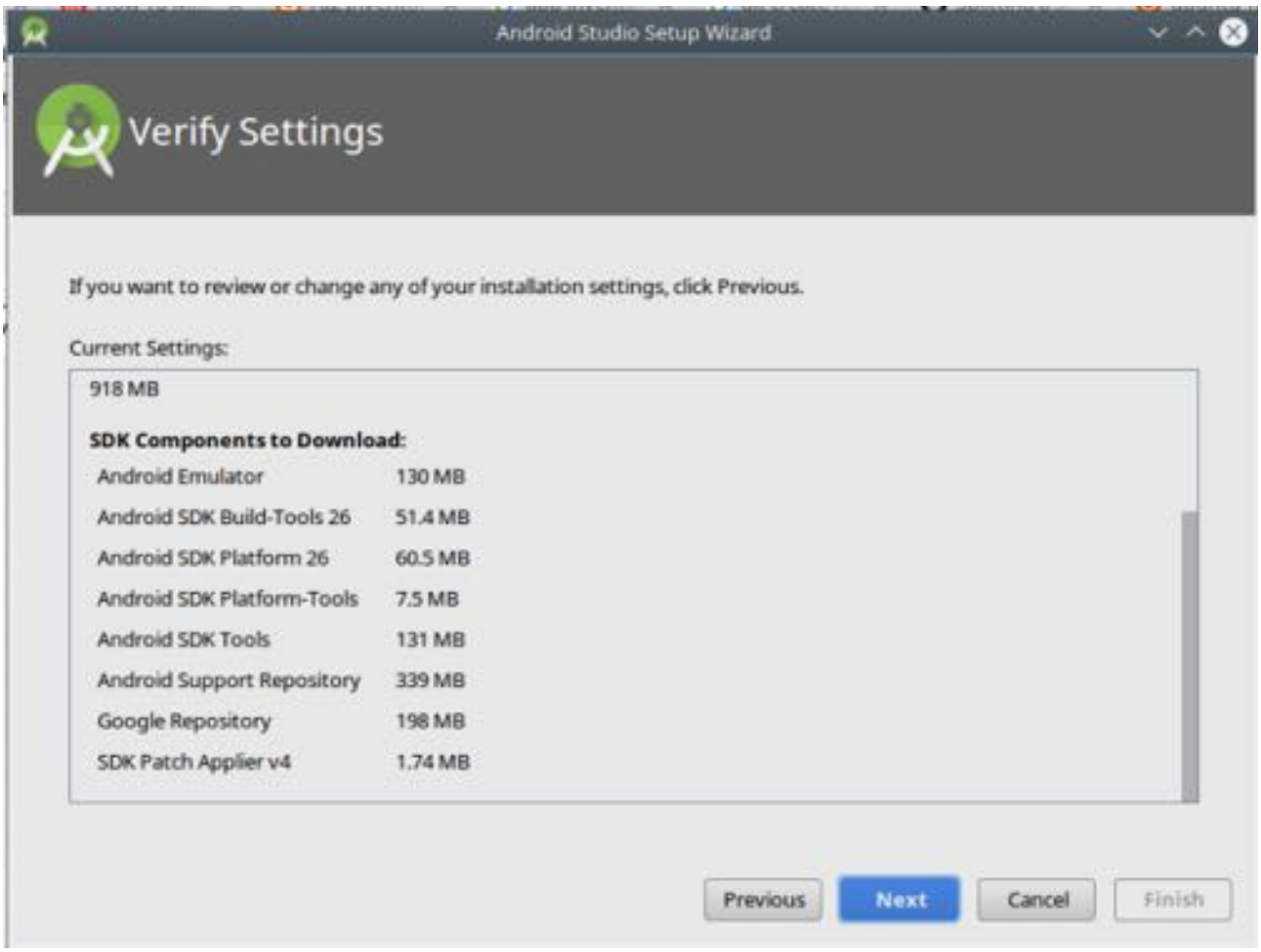


After displaying the license agreement, it will begin downloading and installing the base application. Once it completes and you launch Android Studio, a wizard will lead you through another couple of steps.

First, you'll get a choice on whether you want a "Standard" install, or something custom. Select the Standard install here, it will let you get started quicker.



Then you'll get a message that you need to download some additional components, and it's probably going to take some time.



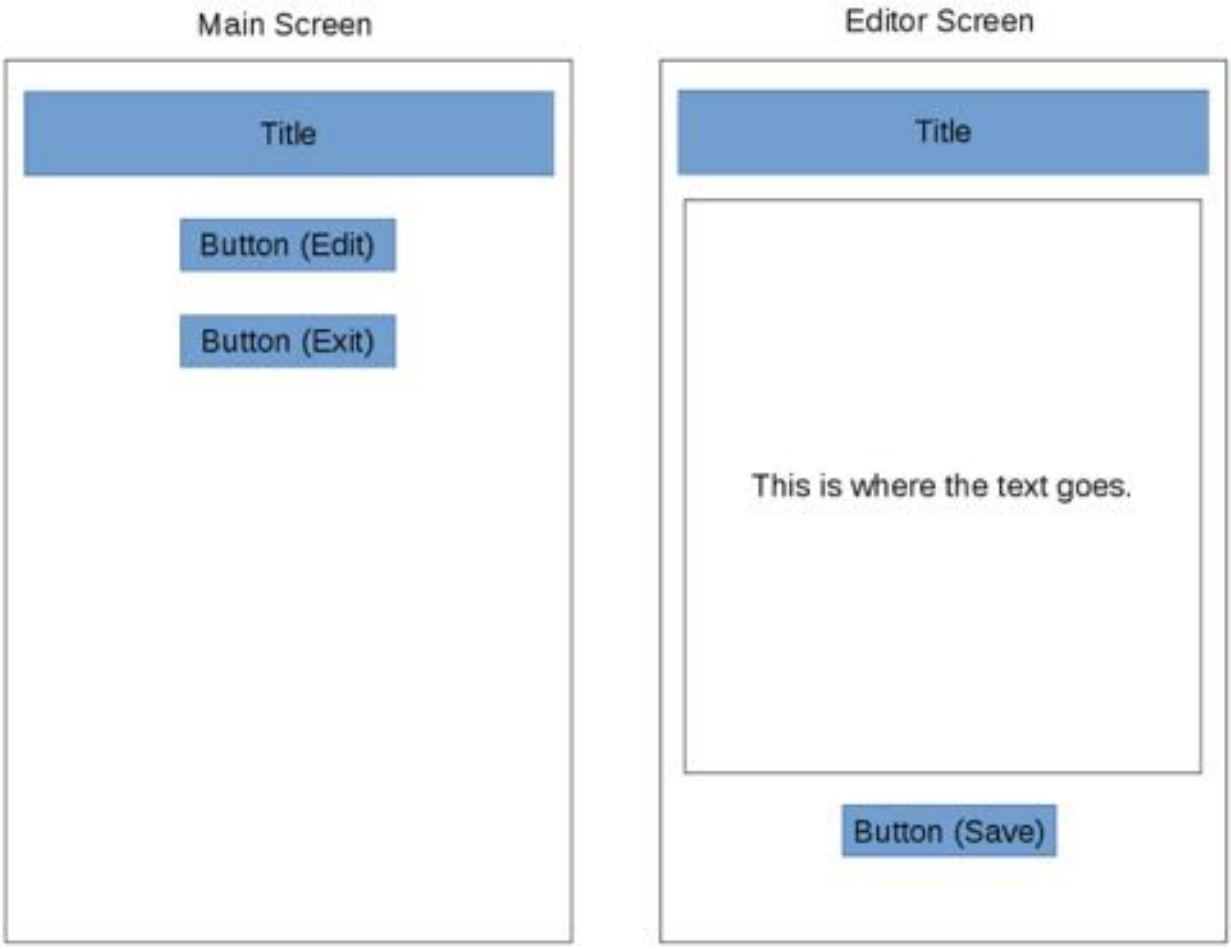
Once everything's installed, you'll get a small splash screen that let's you create a new project, open an existing one, or access your settings.

I know you're ready to get your hands dirty. Without further ado, let's build something.

Building a Simple Android Notepad

Because we have (of course) sat and thought this through before just jumping in, we know our Android app will consist of two screens.

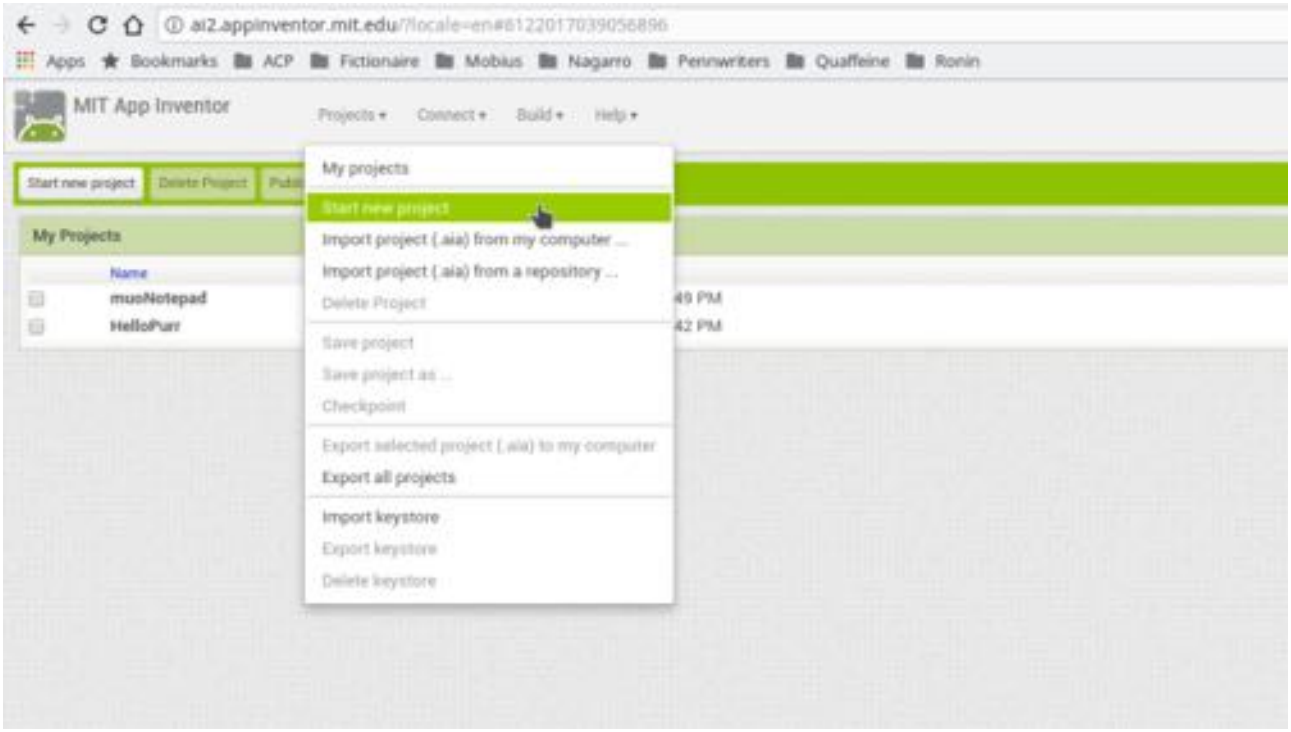
One will allow the user to "edit now" or exit, and the other will do the actual editing. The first screen may seem useless, but it may come in handy later as we add features. The text captured on the "edit" screen will be stashed in a plain text file, because **plain text rules**. The following wireframes give us a good point of reference (and only took 5 minutes to whip up):



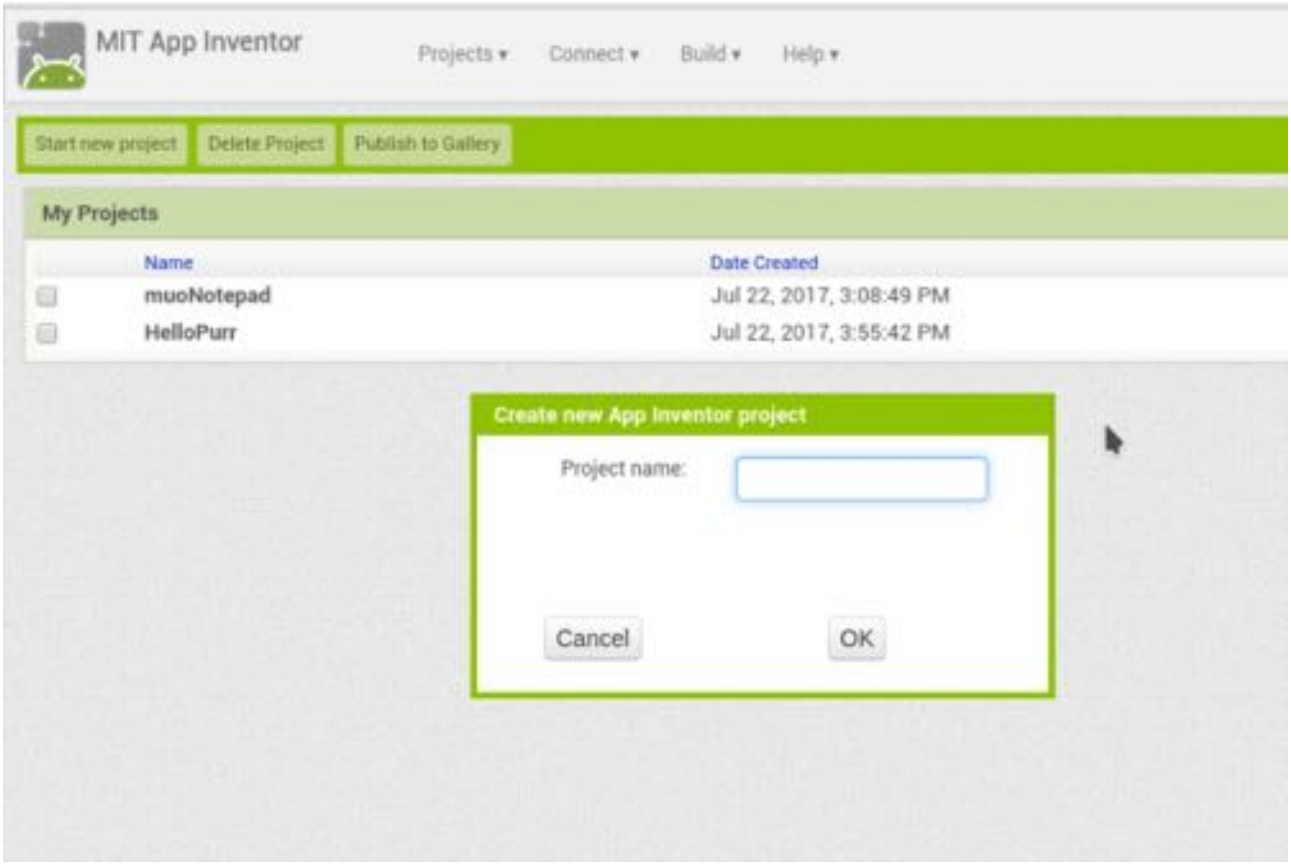
In the next section, we'll build it with MIT's App Inventor.

Getting Started with MIT App Inventor

The first step is to create a new project. Log into App Inventor, then click the **Start new project** button on the left (also available in the **Projects** menu).



You'll get a dialog to give it a name.



But now you're dropped into App Inventor's Designer view, and there's a lot to take in. Let's take a moment to look at each section.

1. The title bar at the top shows your project name (**muoScratchpad**); lets you add, remove, and switch between your app's screens (e.g. **Screen 1**); and toggles between App Inventor's **Designer** and **Blocks** views on the far right.

2. The **Palette** on the left contains all the controls and widgets you'll use. They're divvied up into sections like **User Interface** and **Storage**; we'll use both of these in our app. We'll see how the **Palette** holds different items in the **Blocks** view.

3. The **Viewer** shows you what you're building in WYSIWYG fashion.

4. **Components** is a list of items that are part of the current screen. As you add buttons, text boxes, etc., they'll show up here. Some "hidden" items, like references to files, will show here as well, even though they're not actually part of the user interface.

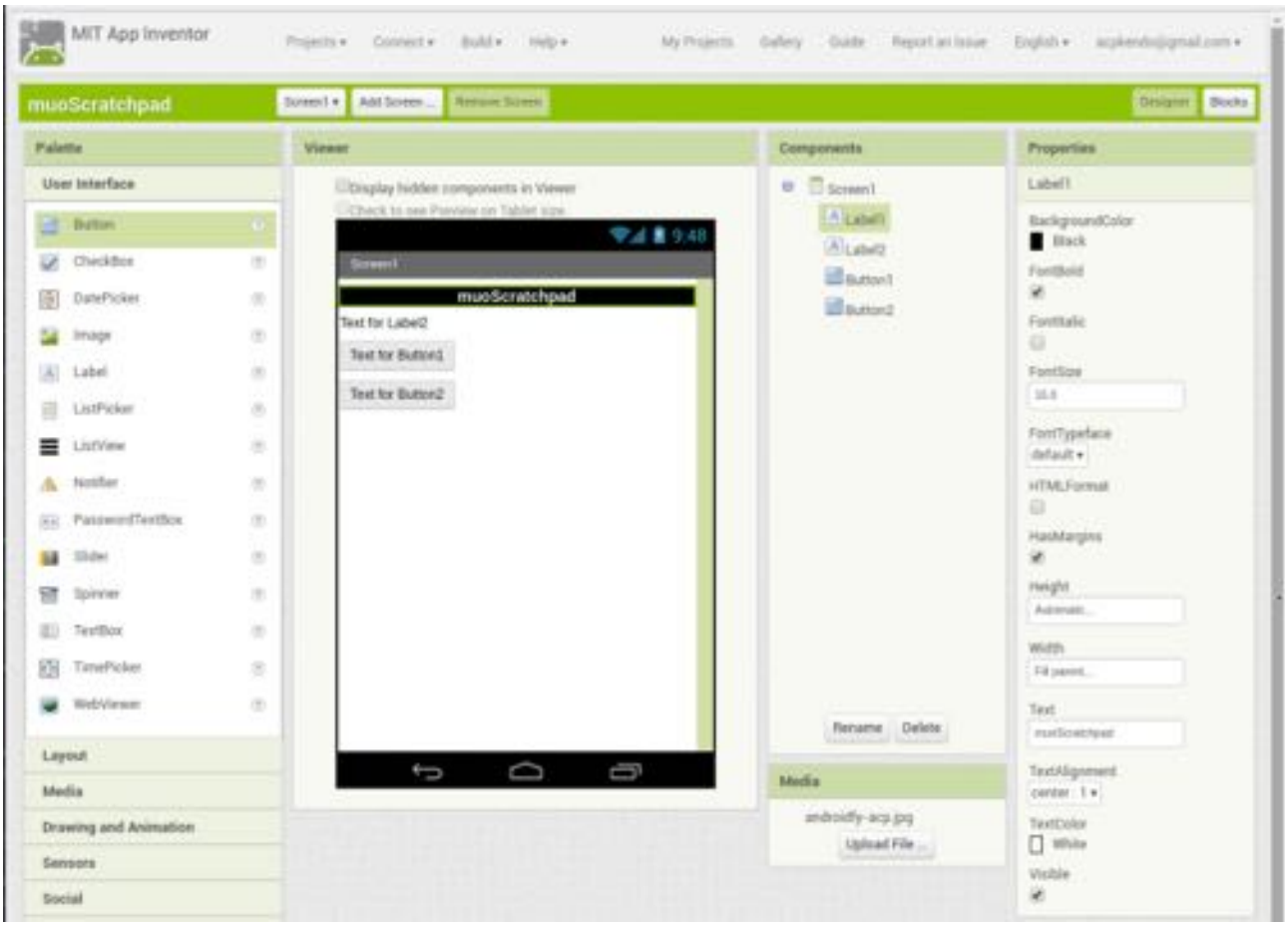
5. The **Media** section lets you upload assets that you'll use in your project, like images or sound clips. (We won't need this one.)

6. Finally, the **Properties** pane lets you configure the currently selected widget. For example, if you're selecting an image widget, you can change its height and width.

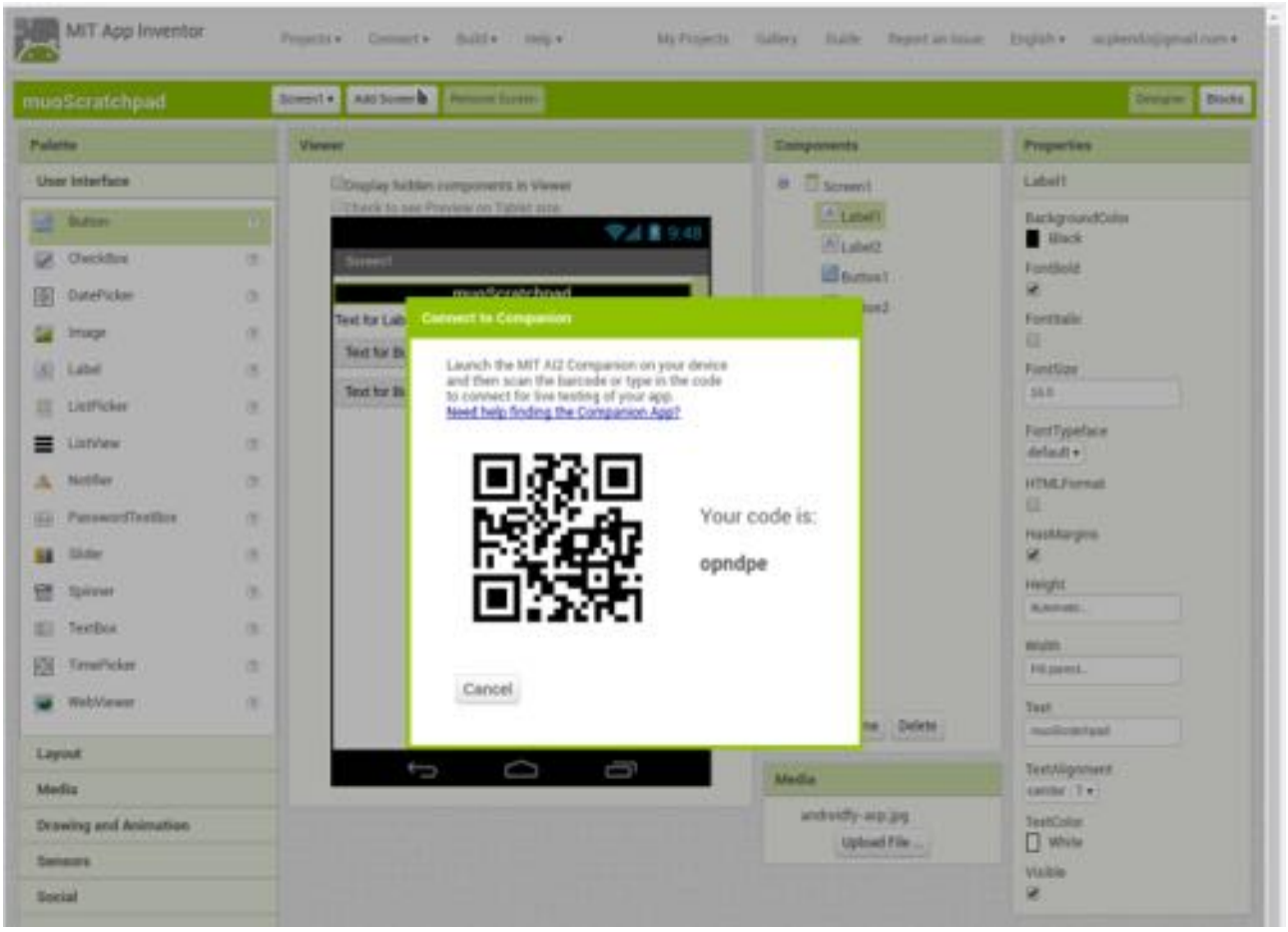
Laying Out Your First Screen: "Main Screen"

Let's put the layout for the "main" screen together in Designer before moving on. Looking at the sketch, we'll need a label for the app name, a line of help text, a button to move to the "edit" screen, and a button to exit. You can see the **User Interface** palette has all the items we need: two **Labels**, and two **Buttons**. Drag these into a vertical column at the top of the screen.

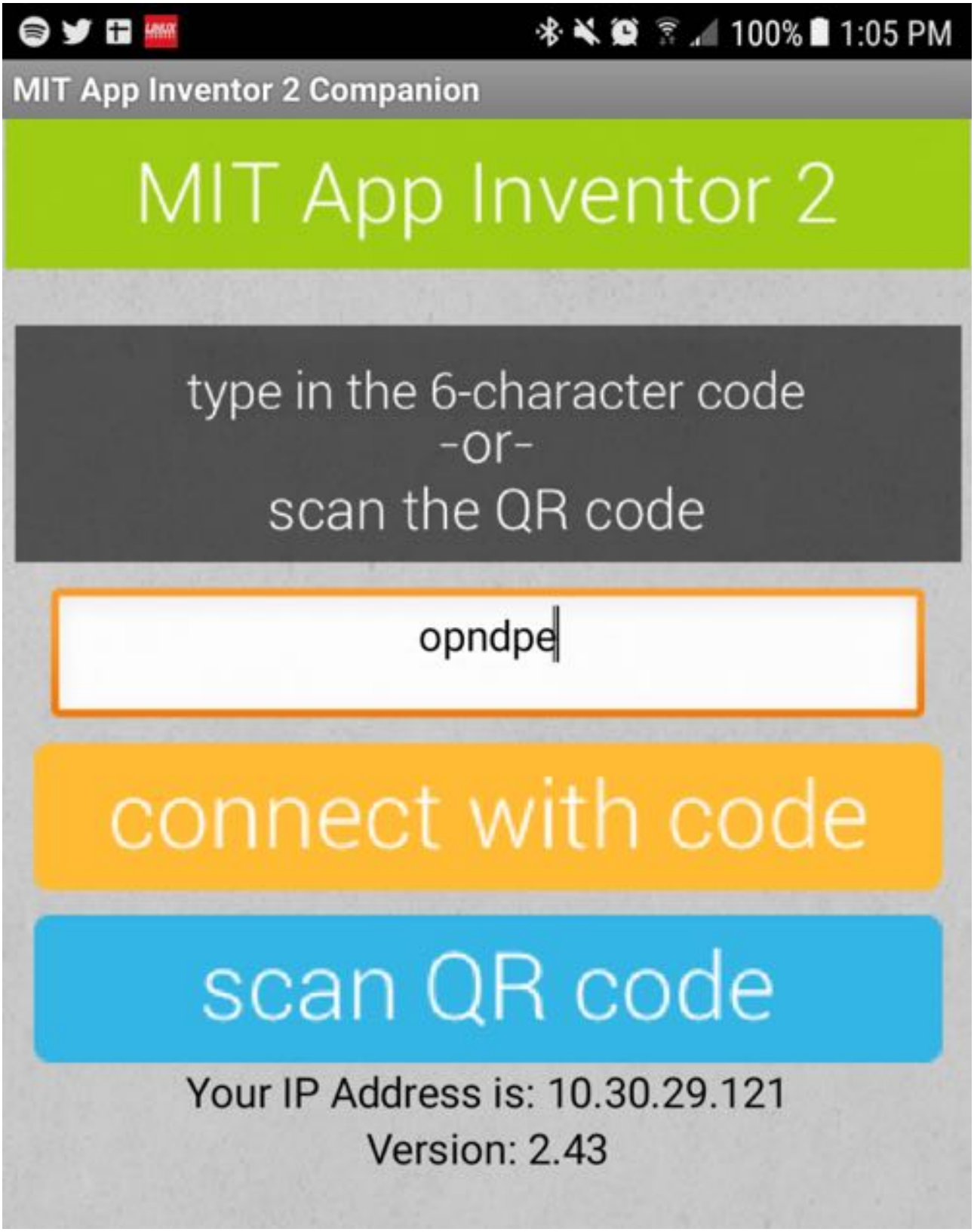
Next we'll configure each one. For the labels, you can set elements like what the text should be, the background color, and alignment. We'll center both of our labels but set the background of the app name to black with white text.



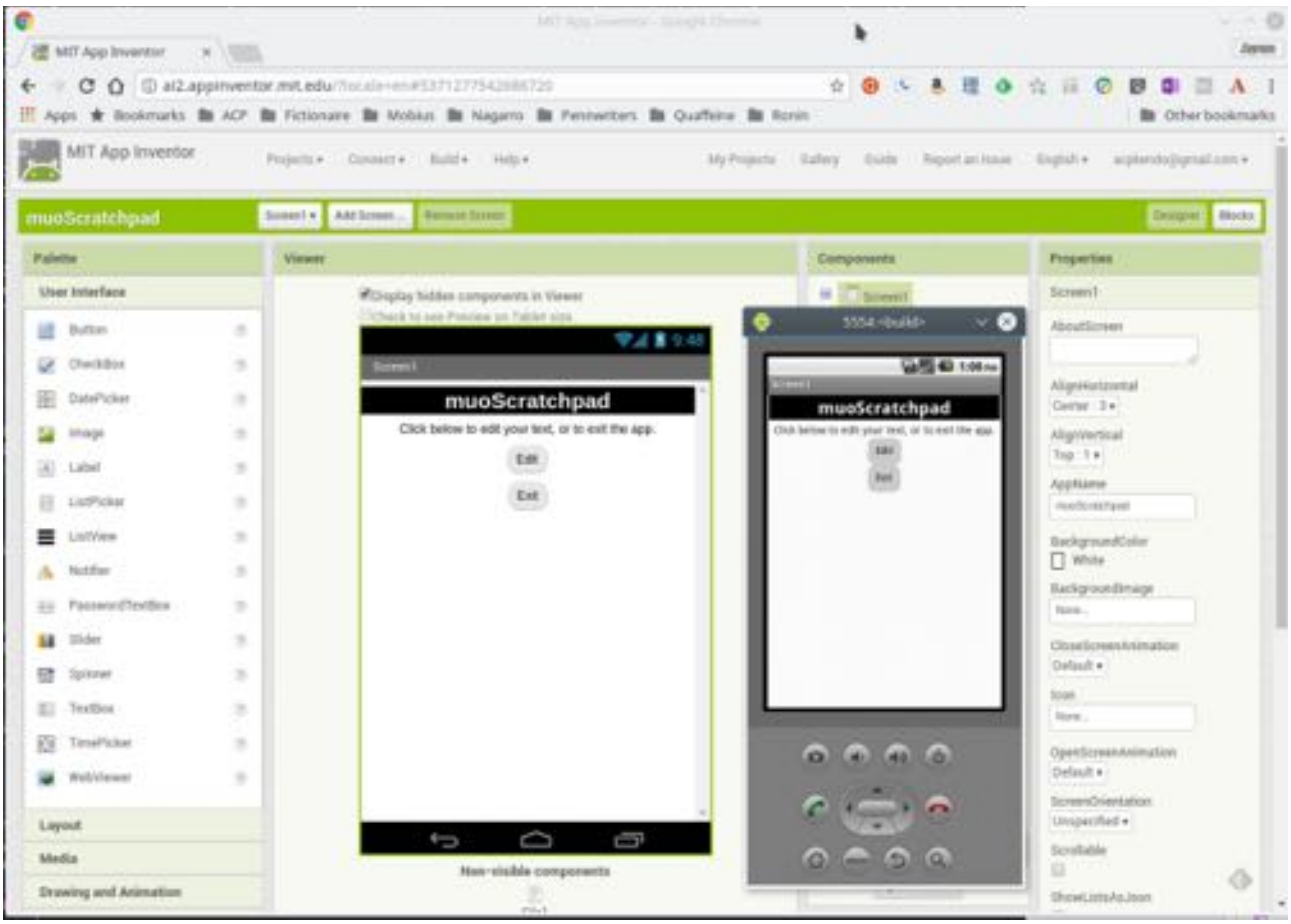
It's time to see how it actually looks on a device. When you're building things, do so in baby steps. **I can't emphasize this enough.**



Don’t build a big list of things into your app in one go, because if something breaks, it takes a **long**time to figure out why. If you’re looking to test on a real phone, you can start up your AI2 Companion app and connect to App Inventor with either the QR code or the six-character code provided.



To preview using the emulator, make sure you've started the aiStarter program described above, then select the **Emulator** item again from the **Connect** menu. Either way, after a short pause, you should see your app pup up, looking something like what you have in the Viewer (the actual layout may depend on the dimensions of your device and emulator).



Since the title looks good, let's change the text on the others as well and align them in the center (this is a property of the screen, **AlignHorizontal**, not the text/buttons). Now you can see one of the really cool aspects of App Inventor — all your changes are done in real time! You can see the text change, the buttons adjust their alignment, etc.

Making it Functional

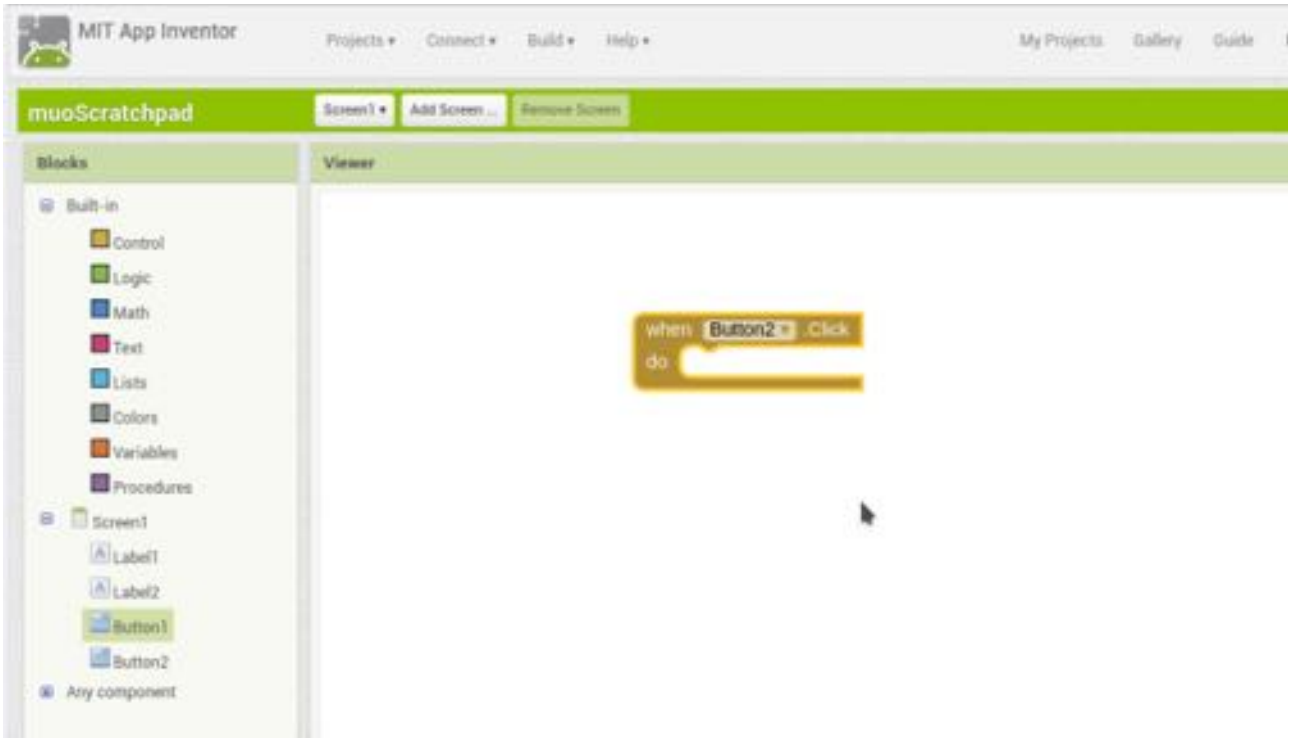
Now that the layout's done, let's add some functionality. Click the **Blocks** button in the upper left. You'll see a similar layout as the Designer view, but you'll have some different choices arranged in categories. These are programming concepts rather than interface controls, but like the other view, you'll use drag-and-drop to put these together as part of your app.

The left-hand Palette contains categories like **Control**, **Text**, and **Variables** in the "Built-in" category. The blocks in this category represent functions that will happen largely behind the scenes, such as the **Math** items that can perform calculations. Below this is a list of the elements in your screen(s), and the blocks available here will affect those elements. For example, clicking on one of our Labels shows blocks that can change that label's text, while the Buttons have blocks to define what happens when you click them.

In addition to their category (represented by color), each block also has a shape that represents its purpose. These can be roughly divided as follows:

- You can think of items with a big gap in middle, such as the "if-then" block shown above, as ones that handle **events**. When something takes place within the app, the other things inside that gap will run.
- Flat blocks with connectors are one of two things. The first are **statements**, which are the equivalent of commands, the items that will fit in the flows above. In the example above, the **make a list** block is a statement, as is **close application**.
- The other option is **expressions**, which differ only slightly from statements. Where a statement might say "set this to '42'", an expression would be something like "add 22 to 20 and give me the result back." In the above, **is in list** is an expression that will evaluate to either true or false. Expressions are also flat blocks, but they likely have a tab on the left side and a notch on the right.
- Lastly, **values** include numbers ("17" and "42" above), strings of text ("Thing 1" and "Thing 2"), or true/false. They typically have a tab on the left only, as they're something you provide to a statement or expression.

You can certainly go through all the guides and tutorials on App Inventor. However, it's designed for you to just start clicking around and (literally) see what fits. On our initial page, we have two items that need attention (the Buttons), so let's see what we can come up with. One of these (Button2) will close the app when clicked. Since this is an interaction with the button. We can check for Button Blocks and find there's one that starts with **when Button2.click** (or when Button 1 is clicked). This is exactly what we want, so we'll drag this onto the Viewer.



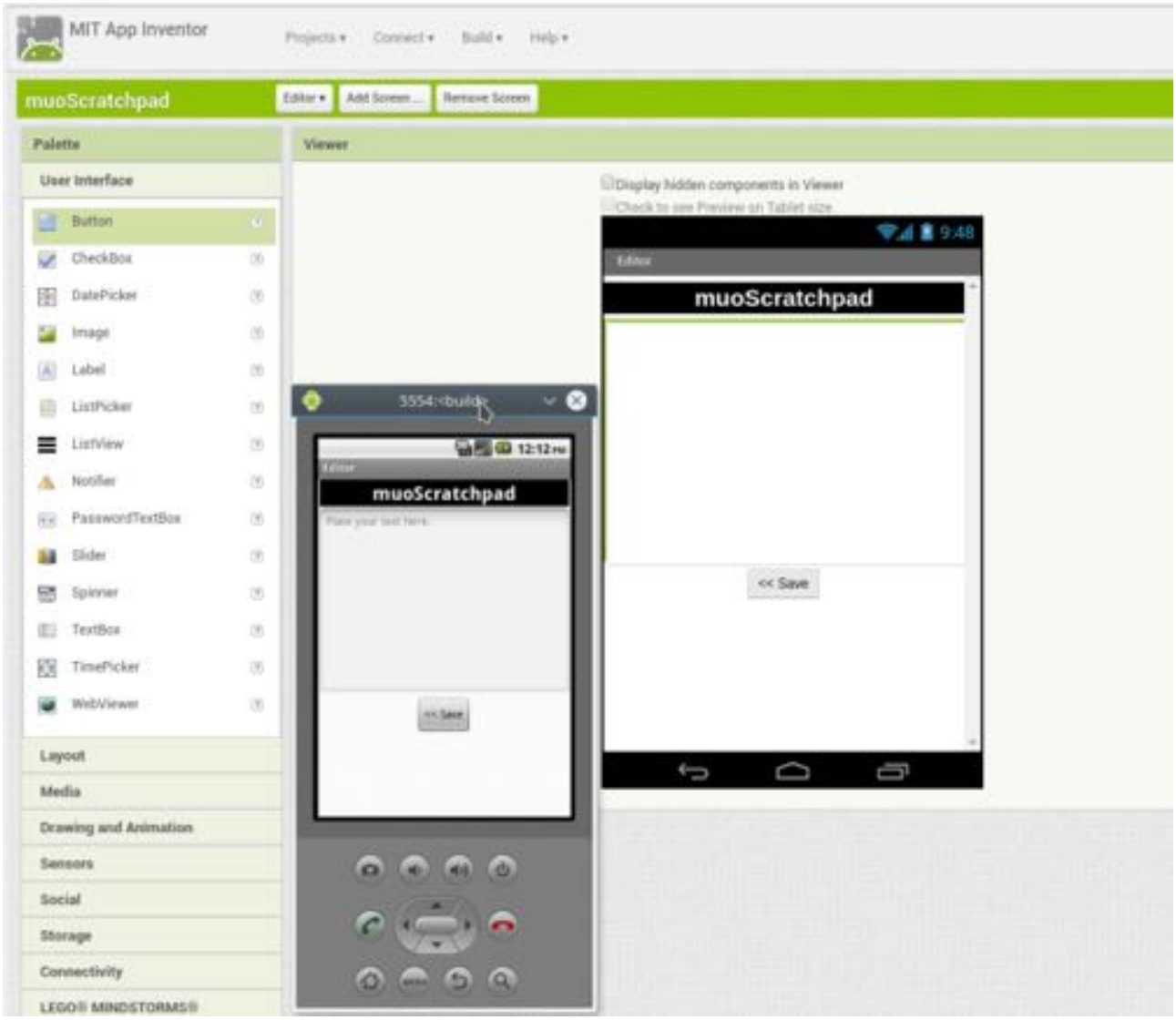
Now when it's clicked, we want the app to close, which sounds like an overall app flow function. Taking a peek in the **Built-in > Control** section, we do indeed see a **close application** Block. And dragging it to the gap in the first block, it clicks into place. Success!

Now when you click the button, the app will close. Let's try it in the emulator. It shows us an error that closing the app isn't supported in the development environment, but seeing this means it works!

Building the Second Screen: Editor Screen

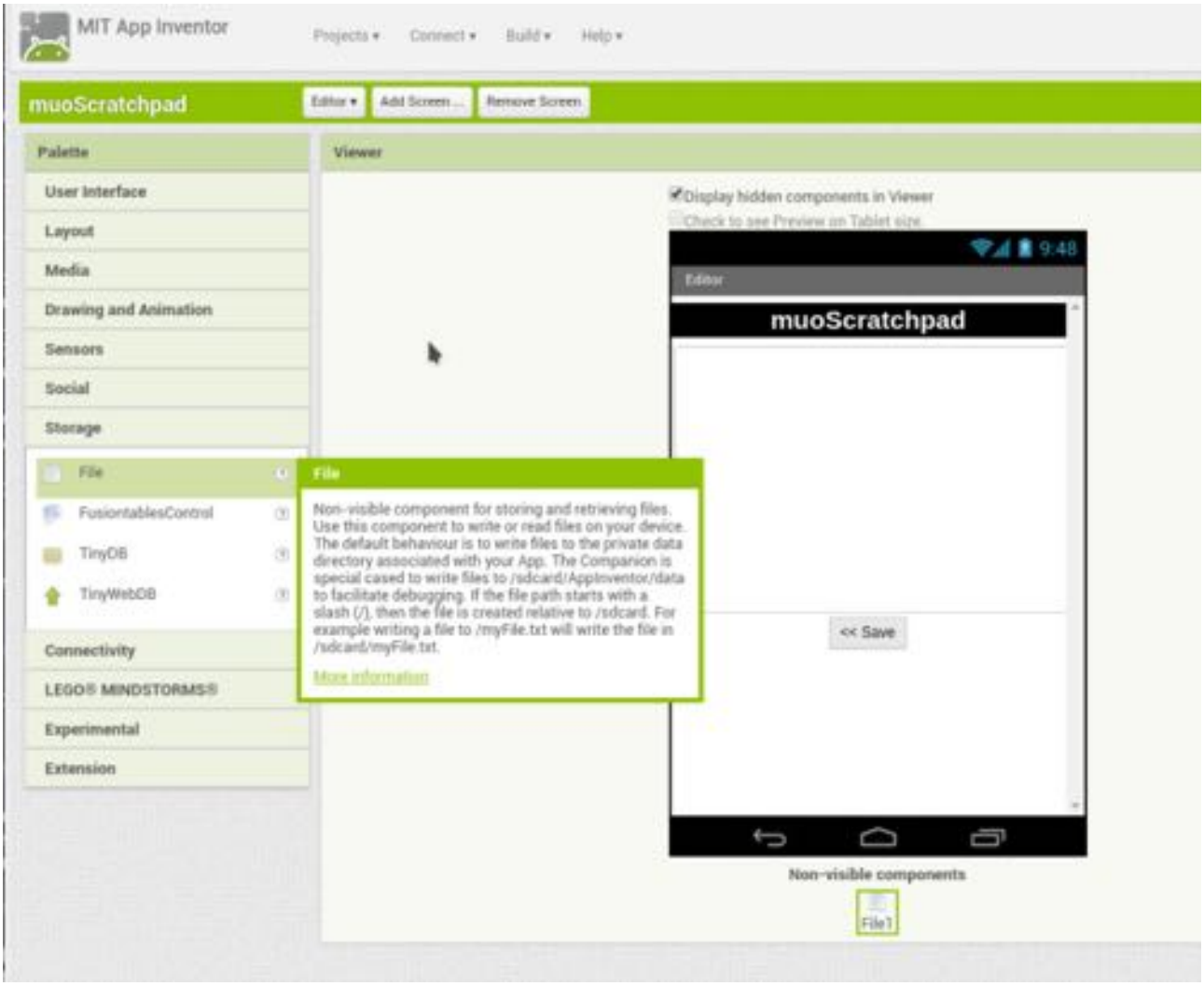
Now let's turn our attention to Button1.

This is supposed to open our editor, so we'd better make sure that editor exists! Let's switch back to the Designer and create a new screen with the same Label as the first screen, a **TextBox** (set to "fill parent" for **Width,** 50% for **Height**, and with **Multiline** enabled) to hold our content, and another Button (labeled "<< Save"). Now check that layout in the emulator!



Before we move ahead, we know we'll want to stash the content from the TextBox, which sounds like **Storage**. Sure enough, there are a couple of options in there.

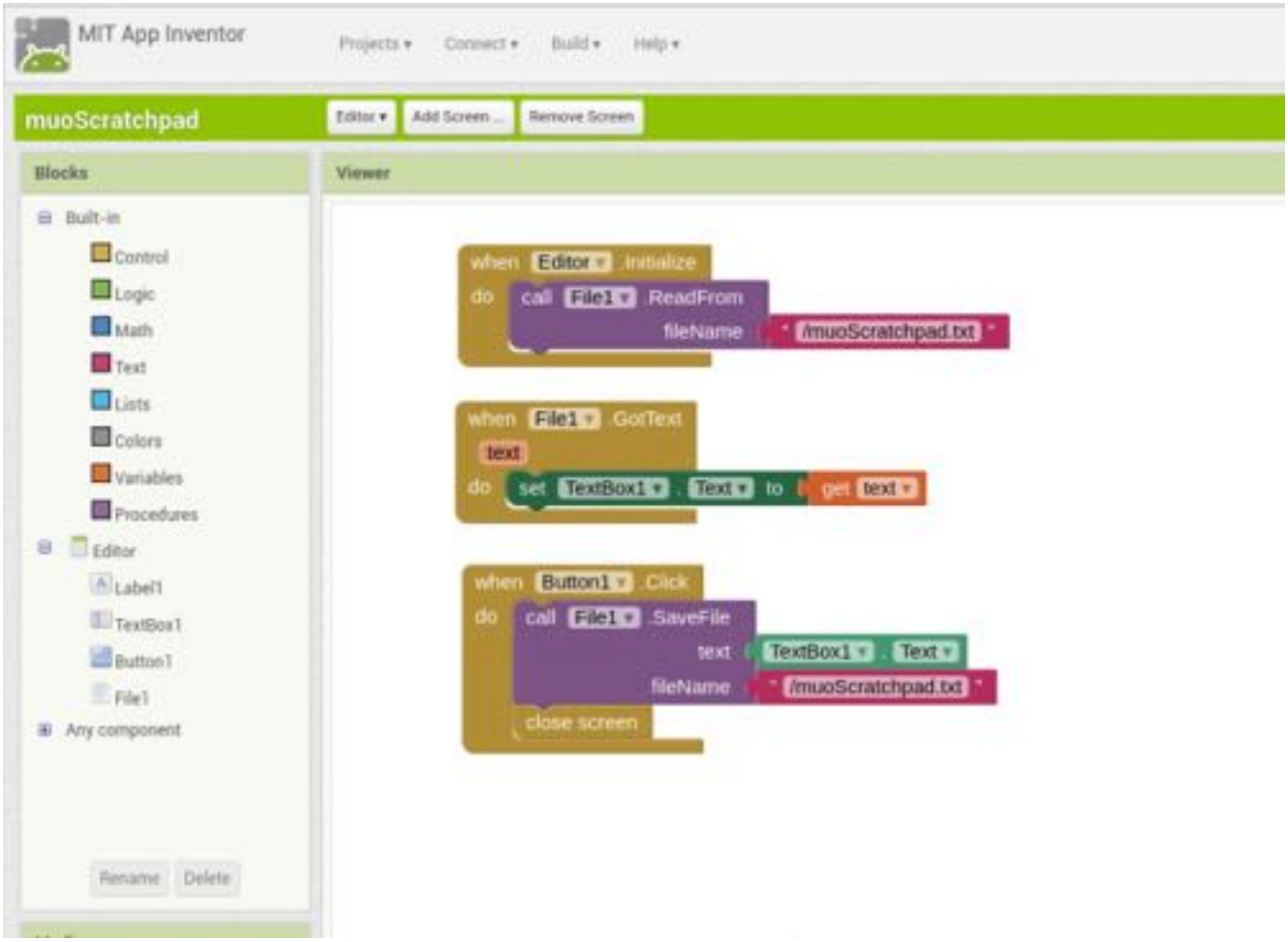
Of these, **File** is the most straightforward, and since we want plain text, it'll be fine. When you put this in the Viewer, you'll notice it doesn't appear. **File** is a **non-visible** component, as it works in the background to save the content to a file on the device. The help text gives you an idea how this works, but if you want these items visible, just check the **Display hidden components in Viewer** checkbox.



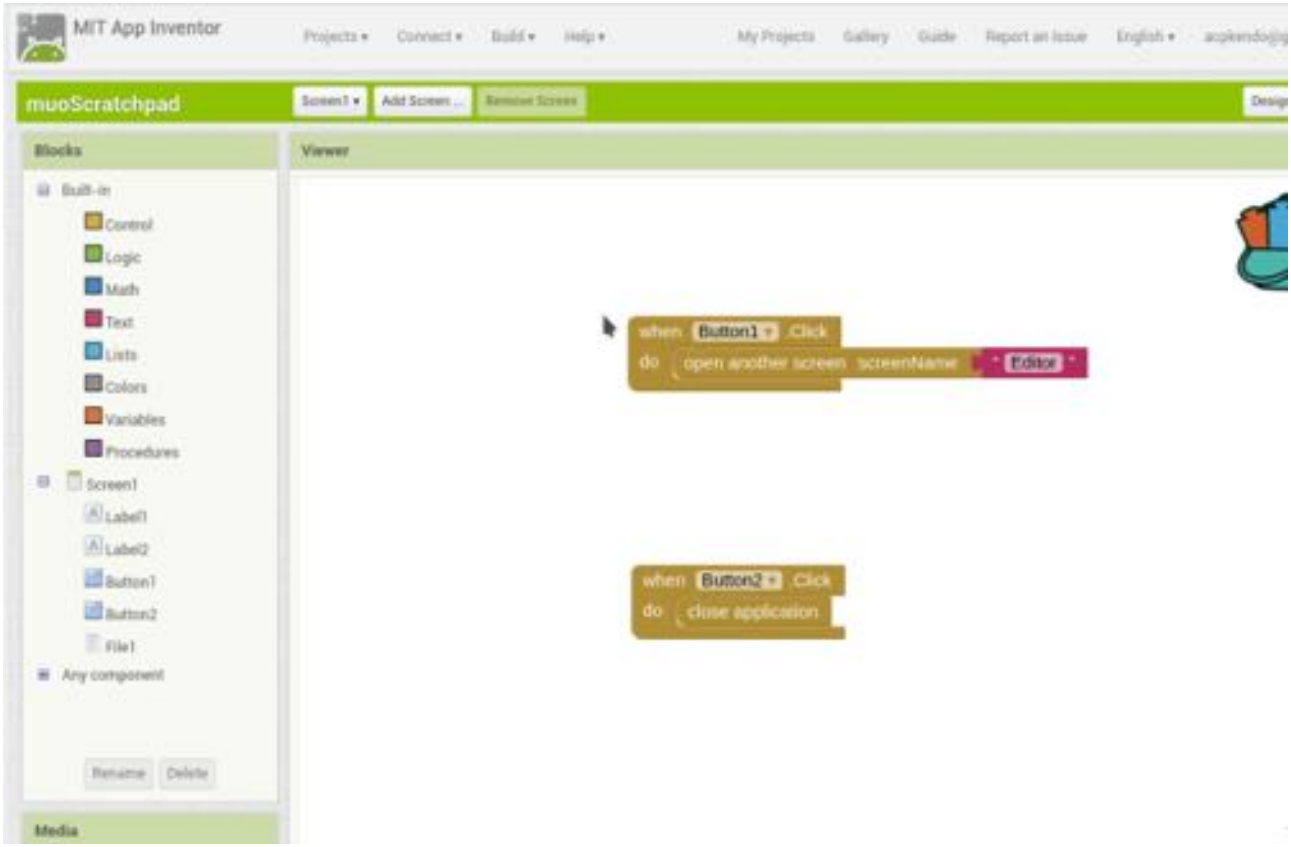
Switch to the Blocks view now — it's time to program. The only behavior we need is when the "<< Save" button is clicked, so we'll grab our **when Button1.click** Block. Here's where App Inventor really starts to shine.

First, we'll save the content of the TextBox by grabbing the **call File1.saveFile** block, and providing it the text we want (using TextBox1's **TextBox1.text**, which retrieves its contents) and a file to store it (just provide a path and file name with a Text Block — the app will create the file for you if it doesn't exist).

Let's also set up the screen to load the contents of this file when it opens (**Editor > when Editor.initialize** Block). It should **call File1.ReadFrom** which points to our filename. We can capture the result of reading the text file using **File > when File1.GotText**, assign that content to the TextBox using the **TextBox > set TextBox.Text to** block, and hand it the **get text** value. Lastly, after saving, we want a click of Button1 to send us back to the main screen (a **close screen** Block).



Last step is to go back to the main screen and program the first button. We want it to send us to the Editor screen, which is a piece of cake with the **Control > open another screen** block, specifying “Editor.”



What Comes Next?

Now that you've got something that works, what comes next? To enhance it of course! App Inventor gives you access to a wide array of Android functionality. Beyond the simple screens we just created, you can add capabilities including media playback, sending texts, or even a live web view to your app.

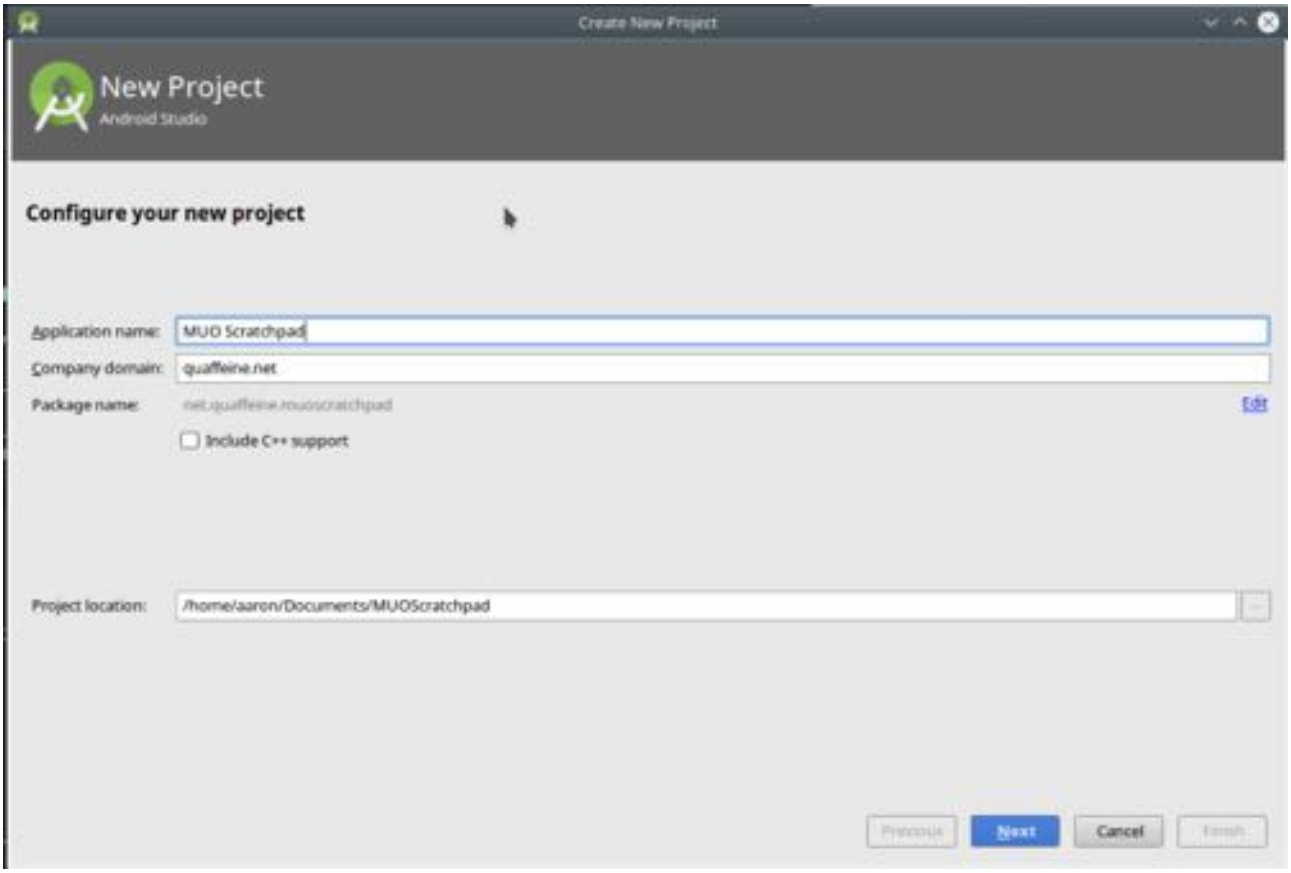
One of the first improvements to come to mind is the ability to select from among multiple files. But a quick **internet search** reveals this requires some supreme hackery in App Inventor. If we want this feature, we'll need to dig into Java and the Android Studio environment.

Development in Java with Android Studio

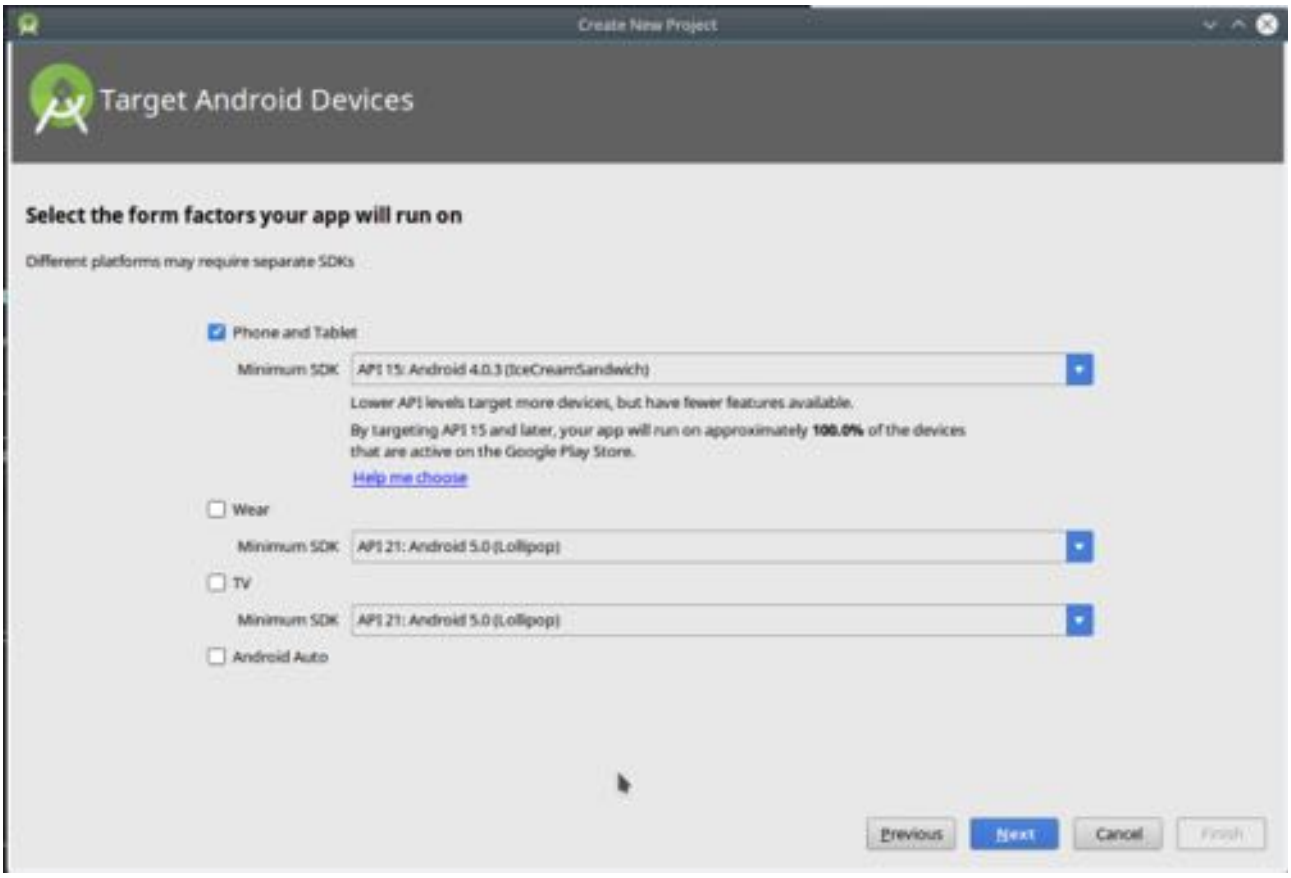
The below sections will describe — at a very high-level — the development of our scratchpad app in Java. It's worth repeating again: while it can pay great dividends down the road, learning Java and Android Studio requires a significant investment of time.

So there won't be as much explanation of **what the code means** below, nor should you worry much about it. Teaching Java is beyond the scope of this article. What we **will do** is examine how close the Java code is to the things we've already built in App Inventor.

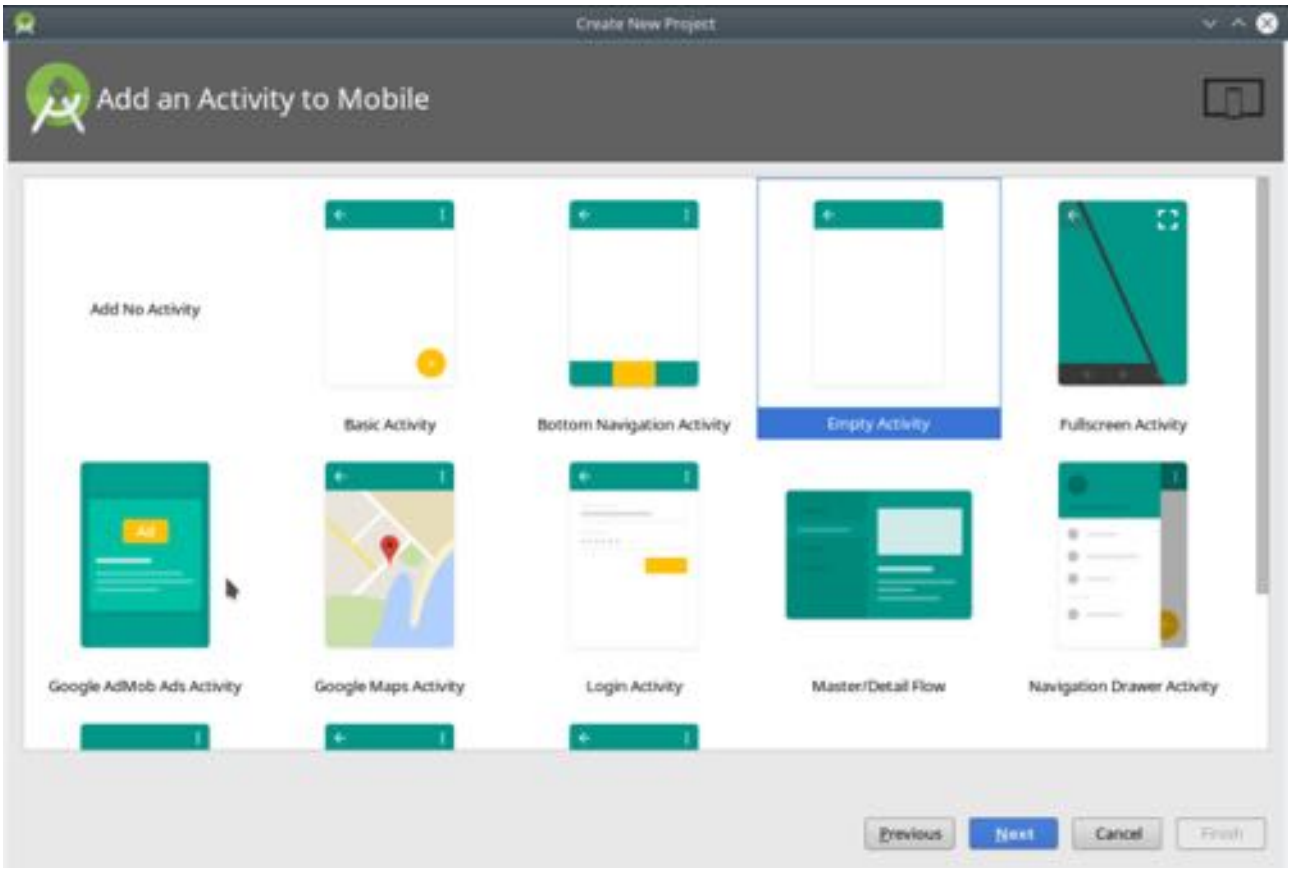
Start by firing up Android Studio, and select the **Start new Android Studio Project** item. You'll be led through a wizard asking a couple things. The first screen asks for a name for your app, your domain (this is important if you submit to the app store, but not if you're just developing for yourself), and a directory for the project.

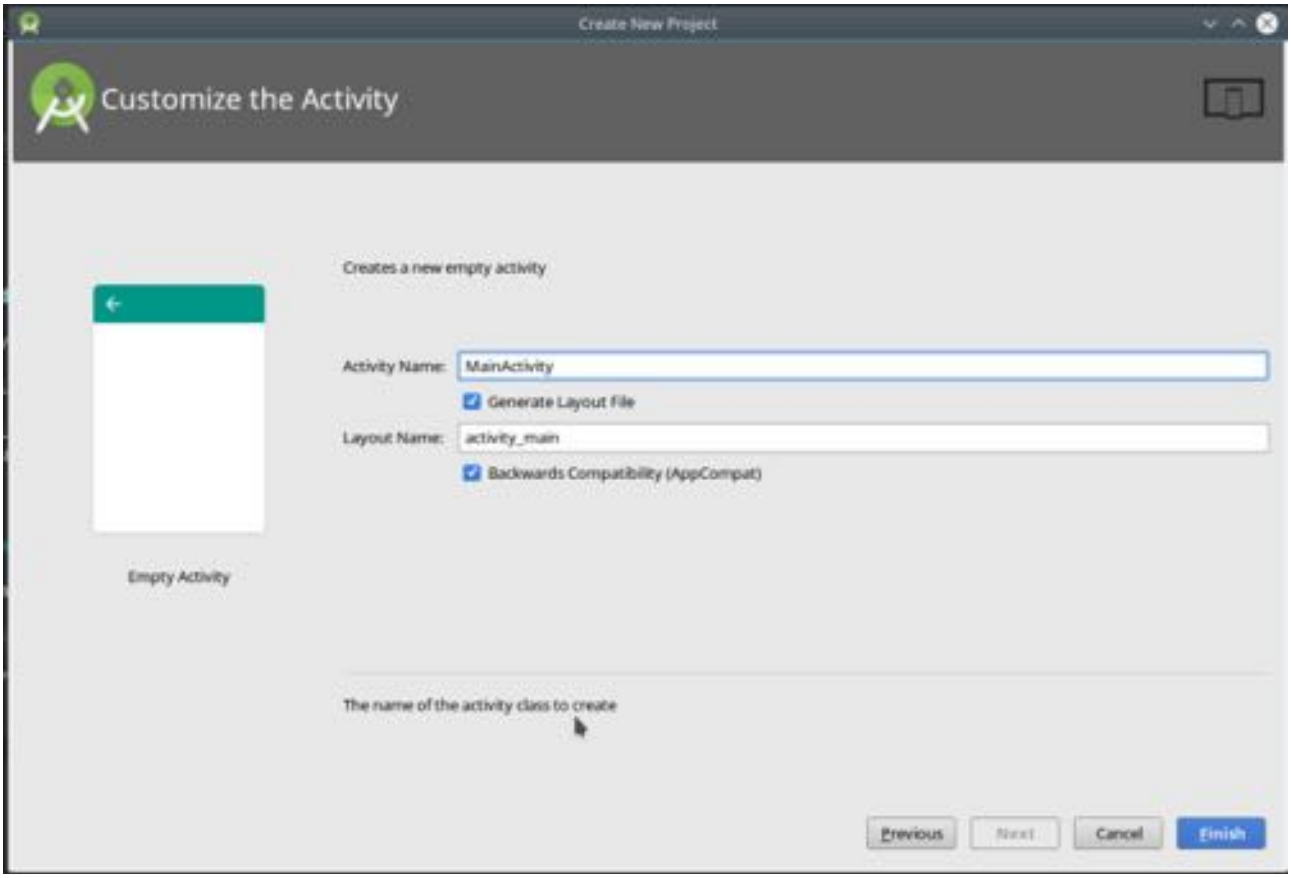


On the next screen, you'll set **the version of Android** to target. Selecting a more recent version will let you include the platform's newer features, but might exclude some users whose devices aren't current. This is a simple app, so we can stick with Ice Cream Sandwich.

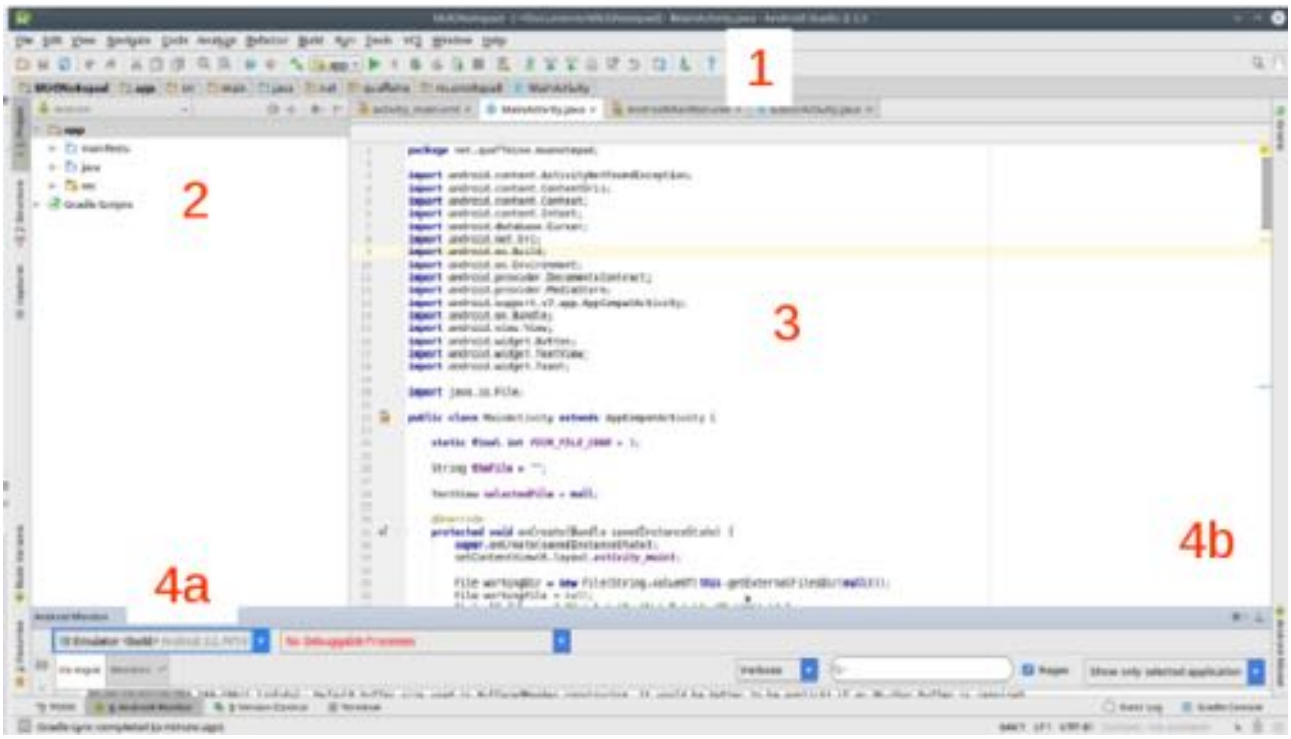


Next we'll select the default **Activity** for our app. Activities are a core concept in Android development, but for our purposes, we can define them as screens. Android Studio has a number you can select from, but we'll just start with a blank one and build it ourselves. The screen after that allows you to give it a name.





Once the new project launches, take a moment to get acquainted with Android Studio.



1. The top toolbar has buttons for a wide variety of functions. The one that's most important for us is the **Run** button, which will build the app and launch it in the emulator. (Go ahead and try it, it'll build just fine.) There are others such as **Save** and **Find**, but these work via the keyboard shortcuts we're all used to (Ctrl+S and Ctrl+F, respectively).

2. The left-hand **Project** pane shows the content of your project. You can double-click on these to open them for editing.

3. The center region is your editor. Depending on what precisely you're editing, this may be text-based or graphical, as we'll see in a moment. This may display other panes as well, such as a right-hand Properties pane (again, like App Inventor).

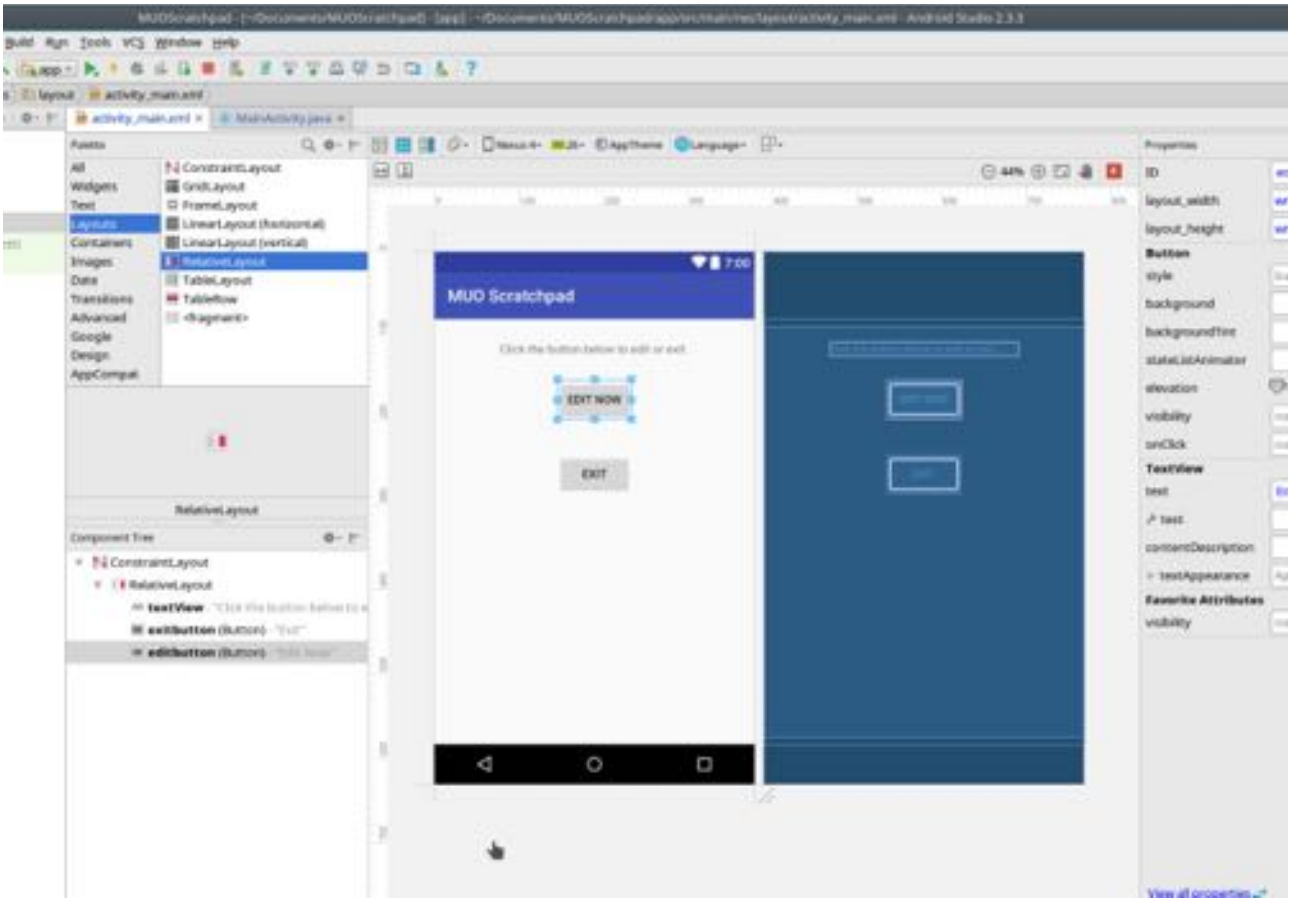
4. The right and bottom borders have a selection of other tools that will pop up as panes when selected. There are things like a terminal for running command line programs and version control, but most of these aren't important for a simple program.

Porting the Main Screen to Java

We'll start by re-building the scratchpad in Java. Looking at our previous app, we can see that for the first screen, we need a label and two buttons.

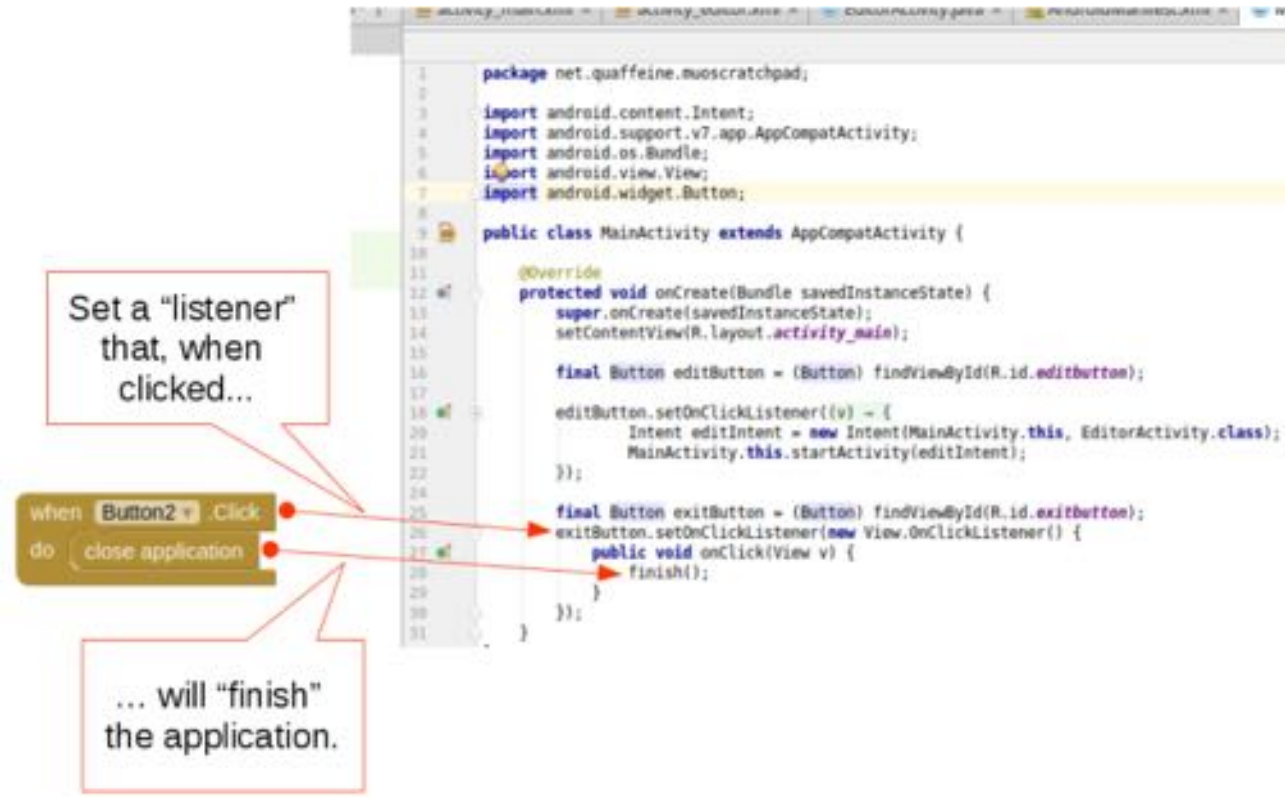
In years past, creating a user interface on Android was a painstaking process involving hand-crafted XML. Nowadays, you do it graphically, just like in App Inventor. Each of our Activities will have a layout file (done in XML), and a code file (JAVA).

Click on the "main_activity.xml" tab, and you'll see the below (very Designer-like) screen. We can use it to drag-and-drop our controls: a **TextView** (like a Label) and two **Buttons**.



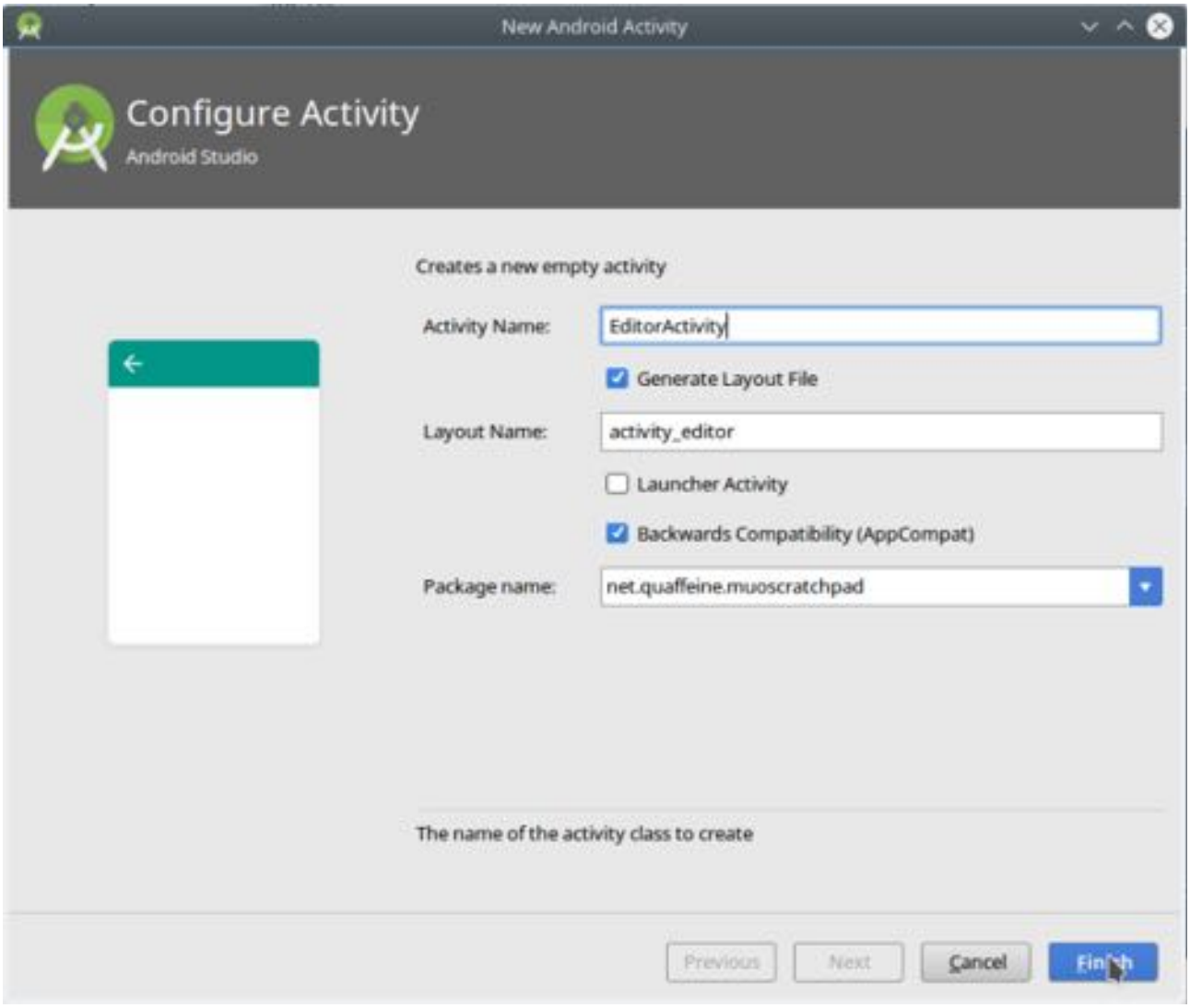
Let's wire up the **Exit** button. We need to create a Button in code as well as graphically, unlike App Inventor that handles that **bookkeeping** for us.

But **like** AI, Android's Java API uses the concept of an "onClickListner." It reacts when a user clicks a button just like our old friend the "when Button1.click" block. We'll use the "finish()" method so that when the user clicks, the app will exit (remember, try this in the emulator when you're done).

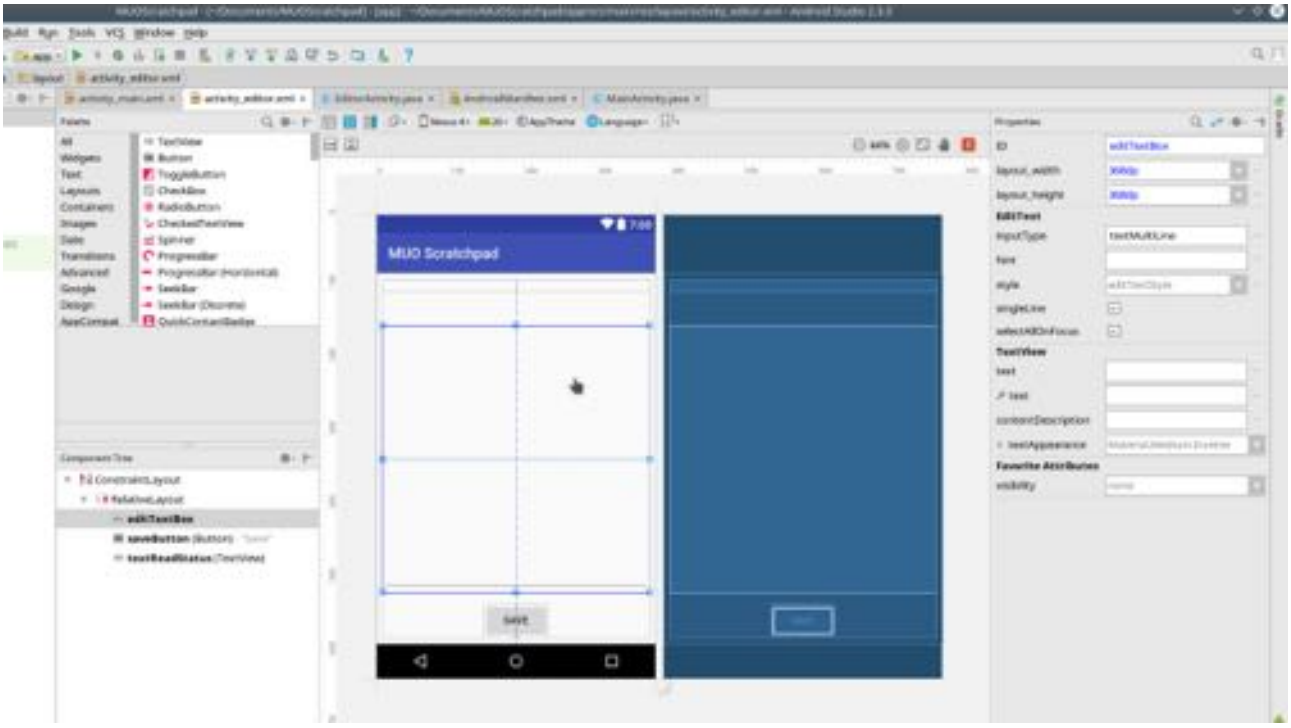


Adding the Editor Screen

Now that we can close the app, we’ll re-trace our steps. Before wiring up the “Edit” Button, let’s make the Editor Activity (screen). Right-click in the **Project** pane and select **New > Activity > Empty Activity** and name it “EditorActivity” to create the new screen.

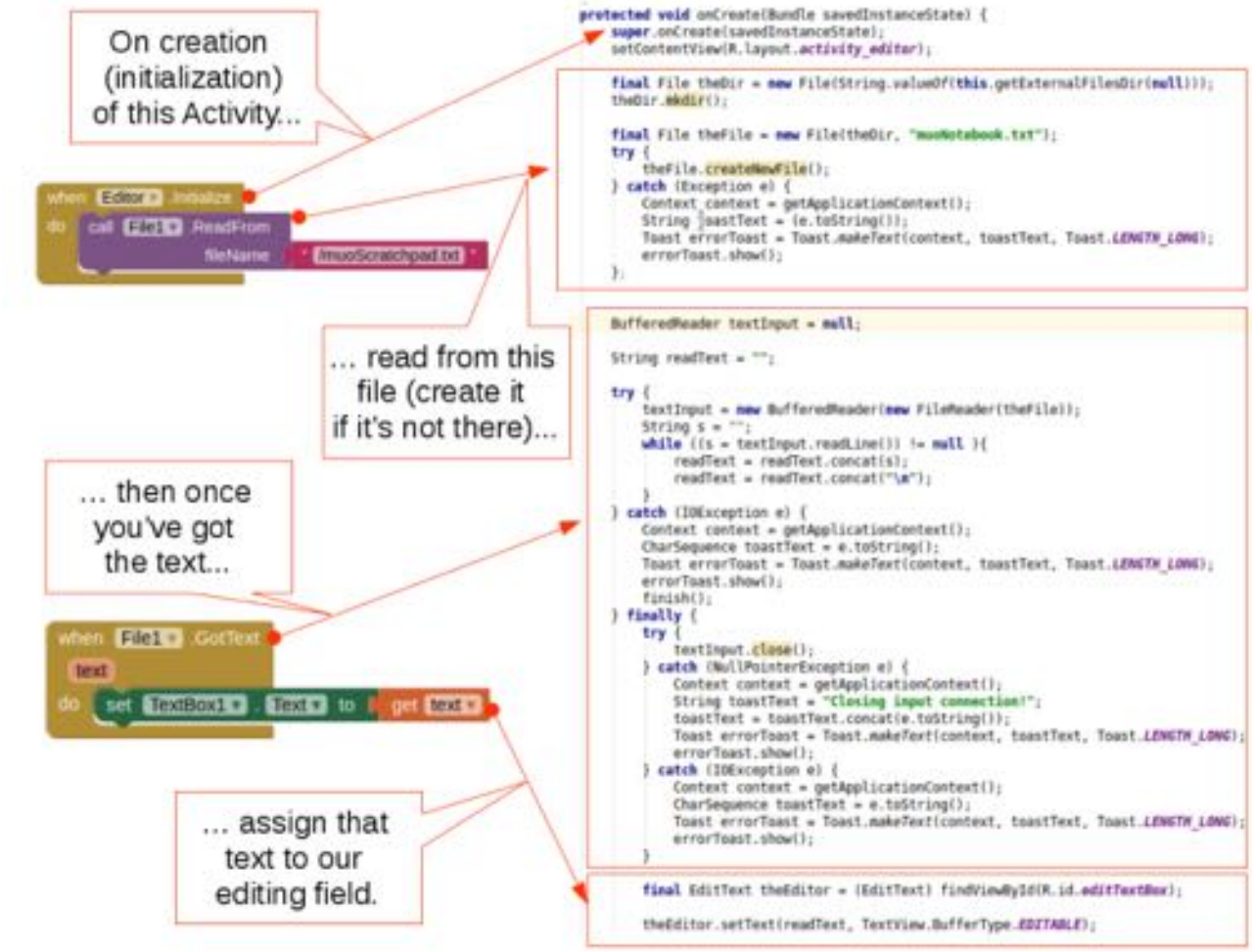


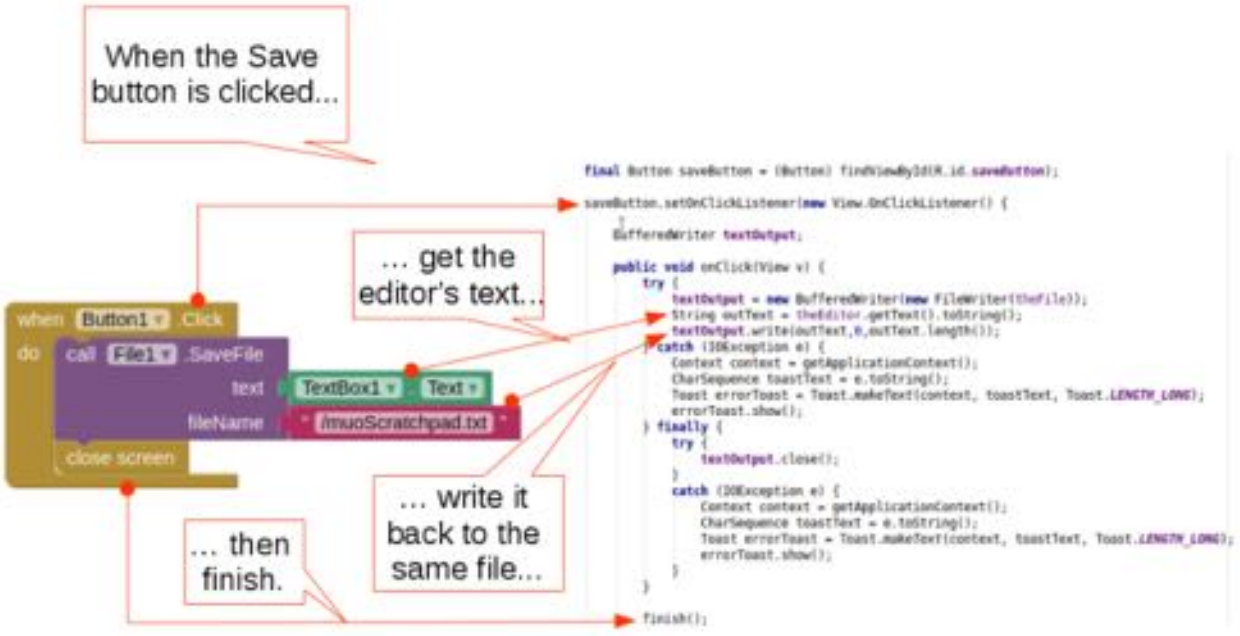
Then we create the layout of the Editor with an **EditTextBox** (where the text will go) and a Button. Adjust the **Properties** of each to your liking.



Now switch to the EditorActivity.java file. We'll code up some similar functions to what we did in App Inventor.

One will create the file to store our text if it doesn't exist, or read its content if it does. A couple of lines will create the **EditTextBox** and load our text into it. Lastly, a bit more code will create the Button and its onClickListener (which will save the text to the file, then close the Activity).





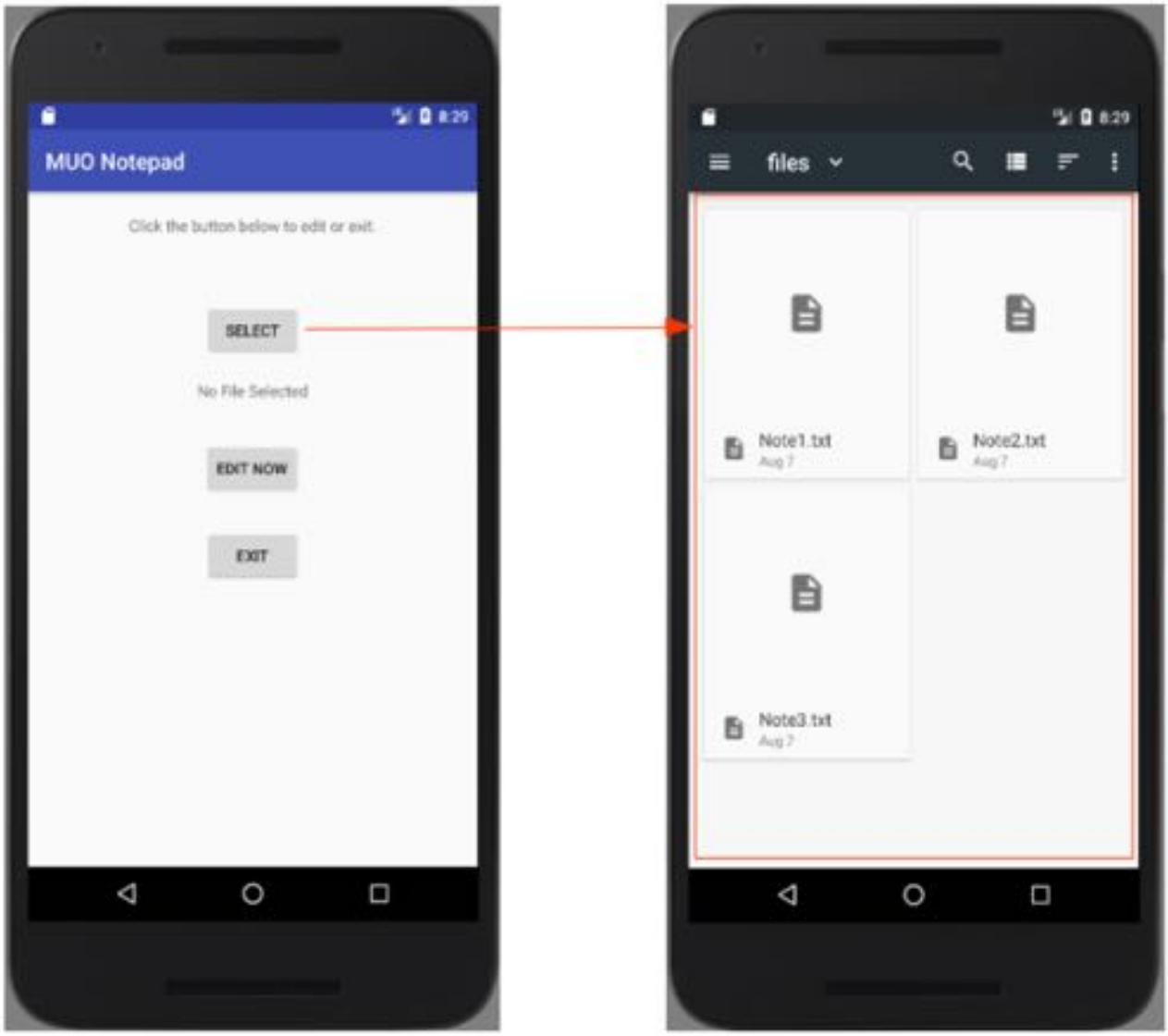
Now when we run it in the emulator, we'll see the following:

1. Prior to running, there is no folder at "/storage/emulated/0/Android/data/[your domain and project name]/files," which is the standard directory for app-specific data.
2. On first run, the Main screen will appear as expected. Still no directory as above, nor our scratchpad file.
3. On clicking the **Edit** button, the directory is created, as is the file.
4. On clicking **Save**, any text entered will be saved to the file. You can confirm by opening the file in a text editor.
5. On clicking **Edit** again, you'll see the previous content. Changing it and clicking **Save** will store it, and clicking **Edit** again will recall it. And so forth.
6. On clicking **Exit**, the app will finish.

Enhancing the App: Select Your Storage File

Now we have a working version of our original App Inventor scratchpad. But we ported it to Java in order to enhance it. Let's include the ability to select from among multiple files in that standard directory. Once we do this, we'll really make this more of a **notepad** than just a scratchpad, so we'll create a copy of the current project **using the instructions here**.

We used an Android Intent to call our Editor Activity from the main one, but they're also a convenient way to call other applications. By adding a couple of lines of code, our Intent will send a request for **file manager applications** to respond. This means we can remove a good portion of the code checking for an creating the file, since the Intent will only allow us to browse/select one that actually exists. In the end, our Editor Activity stays exactly the same.



Getting our Intent to give us back a String (Java text object) that we can pack into our Intent was a challenge. Fortunately, when it comes to programming questions, the internet is your friend. A quick search gives us a couple of options, including code we can paste into our app.

On creation (initialization) of this Activity create some files...

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    File workingDir = new File(String.valueOf(this.getExternalFilesDir(null)));
    File workingFile = null;
    String[] files = {"Note1.txt", "Note2.txt", "Note3.txt"};
    for (String s : files) {
        workingFile = new File(workingDir, s);
        if (!workingFile.exists())
            try {
                workingFile.createNewFile();
            } catch (Exception e) {
                Context context = getApplicationContext();
                CharSequence toastText = e.toString();
                Toast errorToast = Toast.makeText(context, toastText, Toast.LENGTH_LONG);
                errorToast.show();
            }
    }
```

... when the "Select" button is clicked, call file managers for a result, then run "onActivityResult...

```
    final Button selectButton = (Button) findViewById(R.id.selectButton);

    selectButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent fileSelectIntent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
            fileSelectIntent.setType("text/plain");
            fileSelectIntent.addCategory(Intent.CATEGORY_OPENABLE);

            try {
                startActivityForResult(fileSelectIntent, PICK_FILE_CODE);
            } catch (ActivityNotFoundException e) {
                Context context = getApplicationContext();
                CharSequence toastText = e.toString();
                Toast errorToast = Toast.makeText(context, toastText, Toast.LENGTH_LONG);
                errorToast.show();
            }
        }
    });
```

... when a result is provided, assign it to a new label...

```
    selectedFile = (TextView) findViewById(R.id.selectedFile);
```

... the editor starts with the same Intent, we're getting the path...

```
        final Button editButton = (Button) findViewById(R.id.editbutton);

        editButton.setOnClickListener((v) -> {
                Intent editIntent = new Intent(MainActivity.this, EditorActivity.class);
                editIntent.putExtra("file path", theFile);
                MainActivity.this.startActivity(editIntent);
        });

        final Button exitButton = (Button) findViewById(R.id.exitbutton);
        exitButton.setOnClickListener((v) -> { finish(); });
    }
```

... while this is called from above, and in turn calls the "getPath() method below.....

```
    public void onActivityResult(int reqCode, int resultCode, Intent filePathResult) {
        if (reqCode == MainActivity.PICK_FILE_CODE && resultCode == RESULT_OK) {
            Uri filePath = filePathResult.getData();
            theFile = getPath(this, filePath);
            selectedFile.setText(theFile);
        }
    }
```

... containing some code we grabbed from StackOverflow – how does it work? Who cares! For now, it gives us the result we want.

```
    /*
     * Get a file path from a Uri. This will get the the path for Storage Access
     * Framework Documents, as well as the _data field for the MediaStore and
     * other file-based ContentProviders.
     *
     * @param context The context.
     * @param uri The Uri to query.
     * @author paulburke
     */
    public static String getPath(final Context context, final Uri uri) {
```

Code courtesy of StackOverflow

And with this small change and a bit of borrowed code, we can use a file browser/manager application on the device to select the file for storing our content. Now that we're in "enhancement mode," it's easy to come up with a couple more useful improvements:

- We can **choose** from among existing files, but for the moment, we removed our facility to **create** them. We'll need a feature for the user to provide a file name, then create and select that file.
- It might be useful to make our app respond to "Share" requests, so you could share a URL from the browser and add it to one of your note files.
- We're dealing with plain text here, but richer content with images and/or formatting is pretty standard in these types of apps.

With the ability to tap into Java, the possibilities are endless!

Distributing Your App

Now that your app is complete, the first question you'll need to ask yourself is whether you want to distribute it at all! Maybe you've created something so personal and customized it seems like it wouldn't be right for anyone else. But I'd urge you not to think that way. You'll likely be surprised how useful it is to others; if nothing else, it's at least a learning experience showing what a new coder can do.

But even if you decide to keep your new creation to yourself, you'll still need some of the steps below in order to actually install it on your device. So let's learn how to package up your app to share in source code form as well as an installable package.

Source Code Distribution

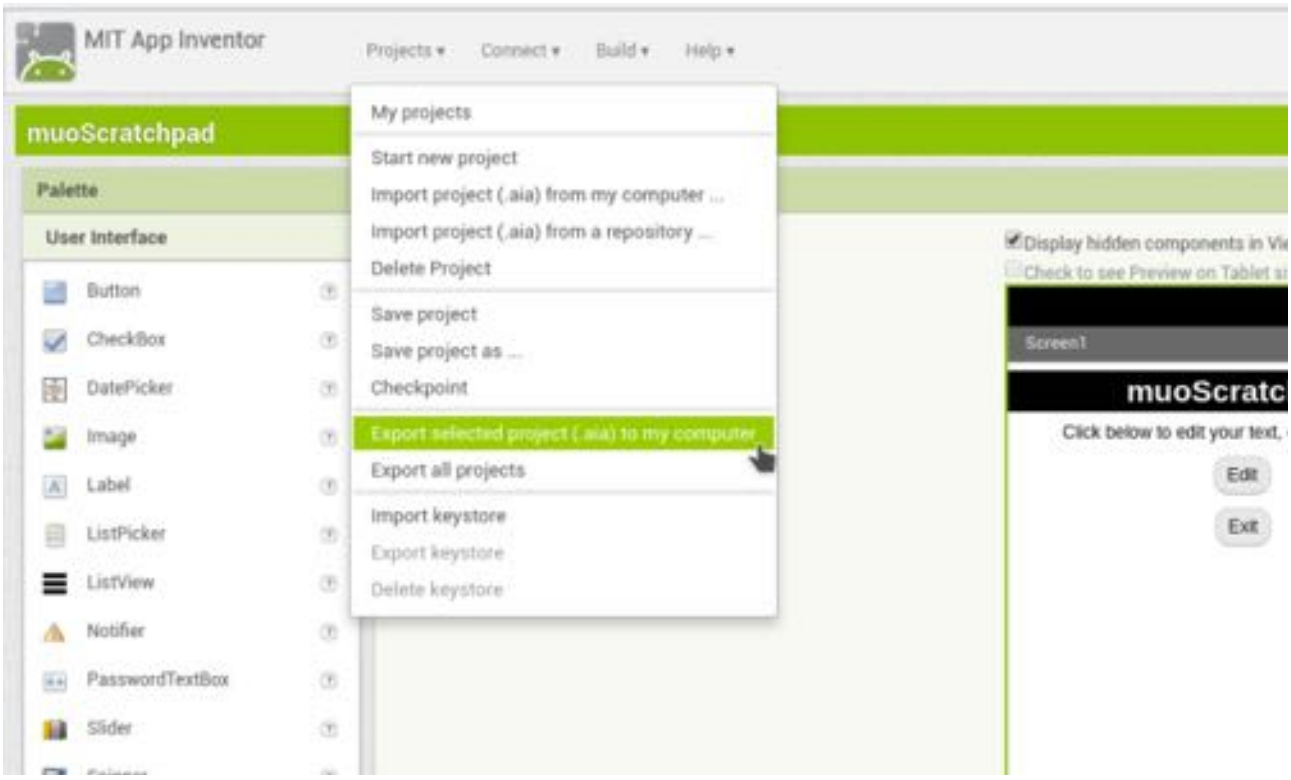
Regardless which method you've been using up to this point, you've been modifying soure code along the way.

While App Inventor does a good job of hiding the actual code behind the scenes, the blocks and UI widgets you've been moving around all represent code. And source code is a perfectly valid way of distributing software, as the open source community can well attest. This is also a great way to get others involved in your application, as they can take what you've done and build on it.

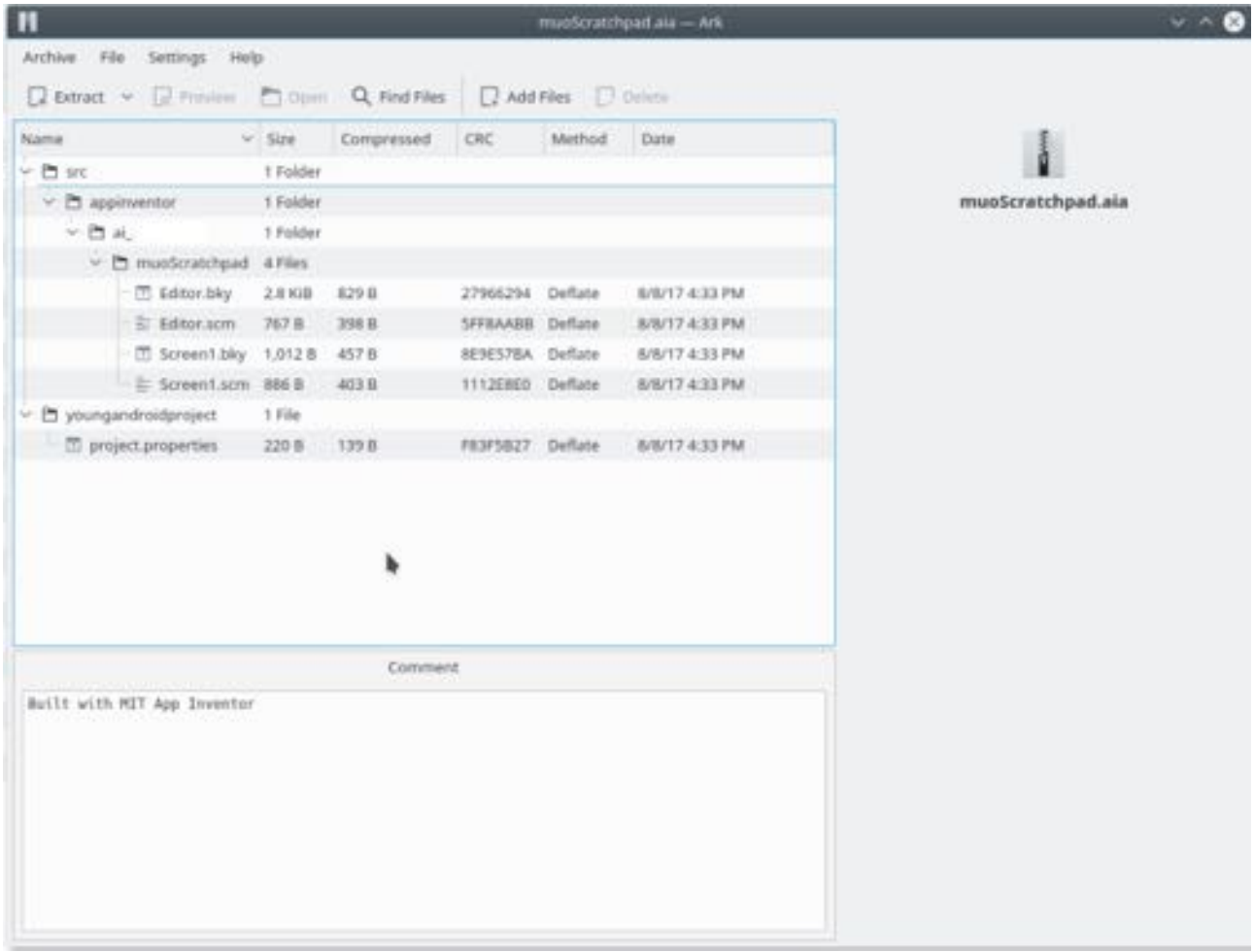
We'll get the source code from both environments in a structured format. Then either someone (ourselves included) can easily import it back into the same program and get up and running quickly.

Exporting Source from App Inventor

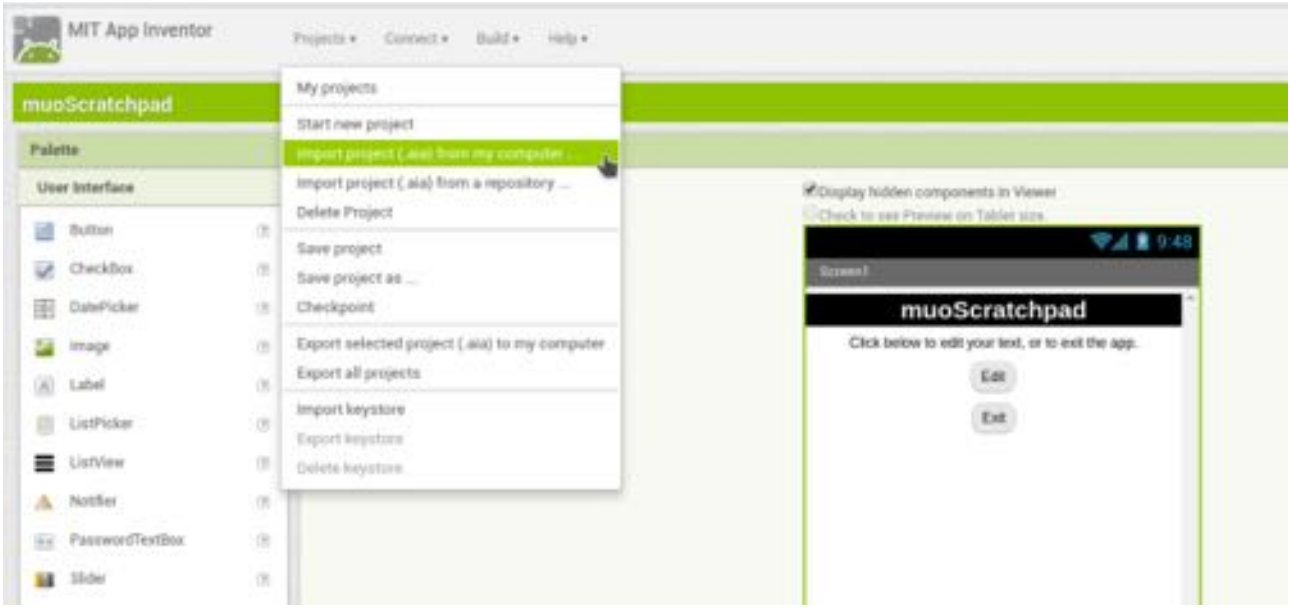
To export from App Inventor, it's a simple matter of opening your project, then from the **Projects** menu, select the **Export selected project (.aia) to my computer**.



This will download the aforementioned .AIA file (presumably “App Inventor Archive”). But this is in fact a ZIP file; try opening it in your favorite archive manager to inspect its contents.



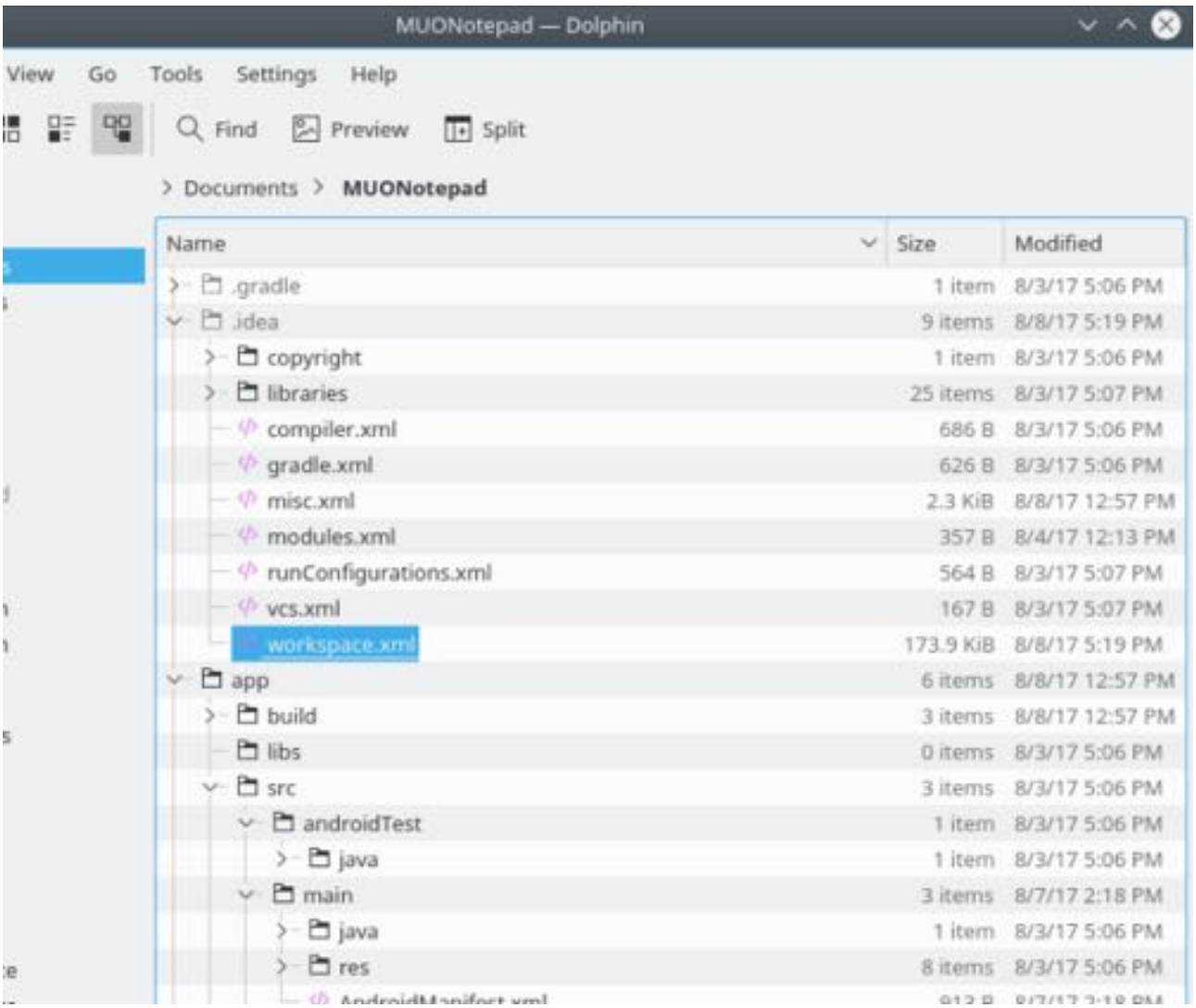
Notice that the contents of the **appinventor/ai_[your user id]/[project name]** folder are a SCM and BKY file. This isn't the JAVA source we saw in Android Studio, so you won't be able to open these up in any old development environment and compile them. However, you (or someone else) can re-import them into App Inventor.



Archiving Source from Android Studio

Getting your Android Studio project out in an archive format is as easy as compressing the project's folder. Then move it to a new location, and open it from the usual **File > Open** item in the main menu.

Android Studio will read your project's settings (**workspace.xml**) and everything should be as it was before.



It's worth noting that archiving that entire folder **will** include some cruft, specifically the files from your program's last build.

These will be cleared and regenerated during the next build, so they're not necessary to keep the integrity of your project. But they don't hurt it either, and it's easier (especially for beginning developers) not to start mucking about with which folders should come along and which shouldn't. Better to take the whole thing rather than miss something you need later.

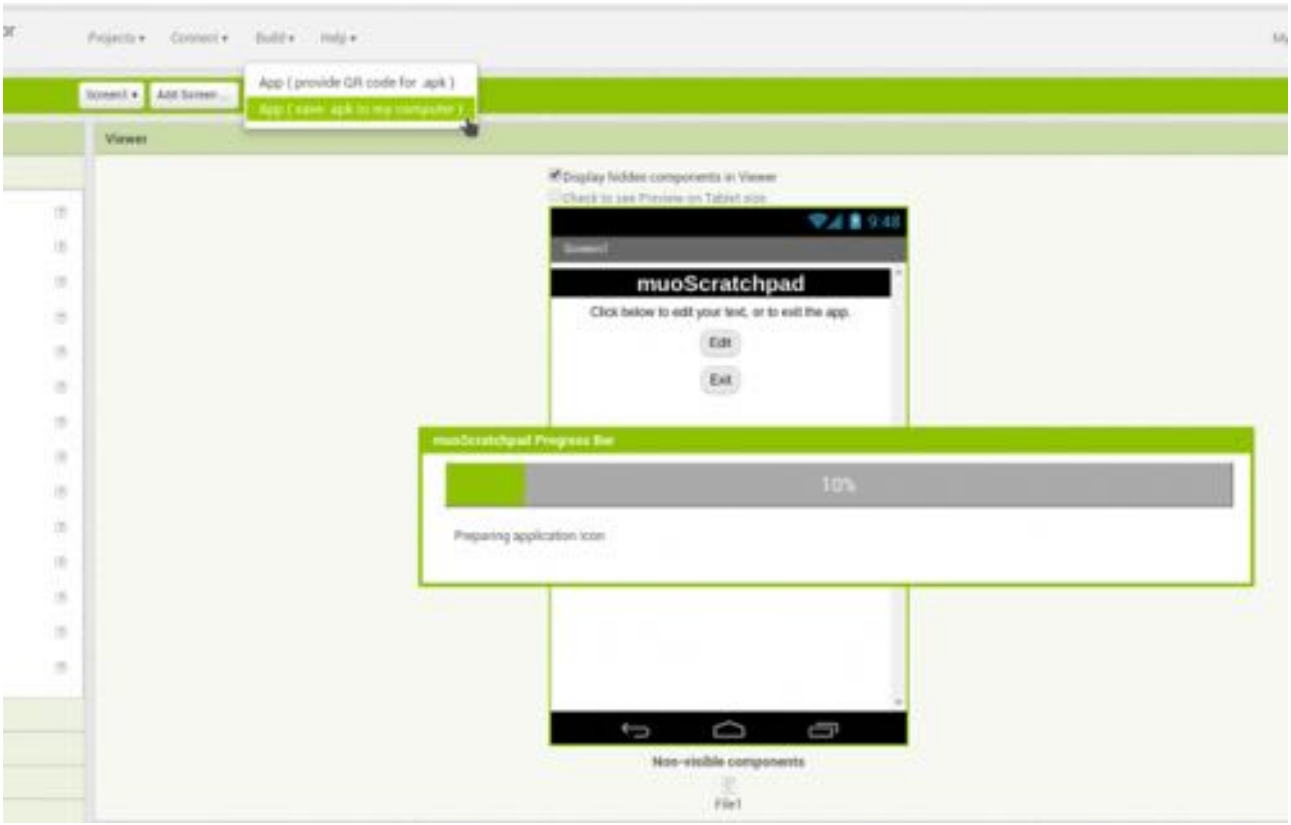
Android Package Distribution

If you want to give a copy of your app to someone just to try it out, an APK file is your best bet. The standard Android package format should be familiar to those who have gone outside the Play Store to get software.

Getting these are just as easy as archiving the source in both programs. Then you can post it on a website (such as F-Droid), or hand it off to some friendly folks to get their feedback. This makes for a great beta test for apps you mean to sell later.

Building an APK in App Inventor

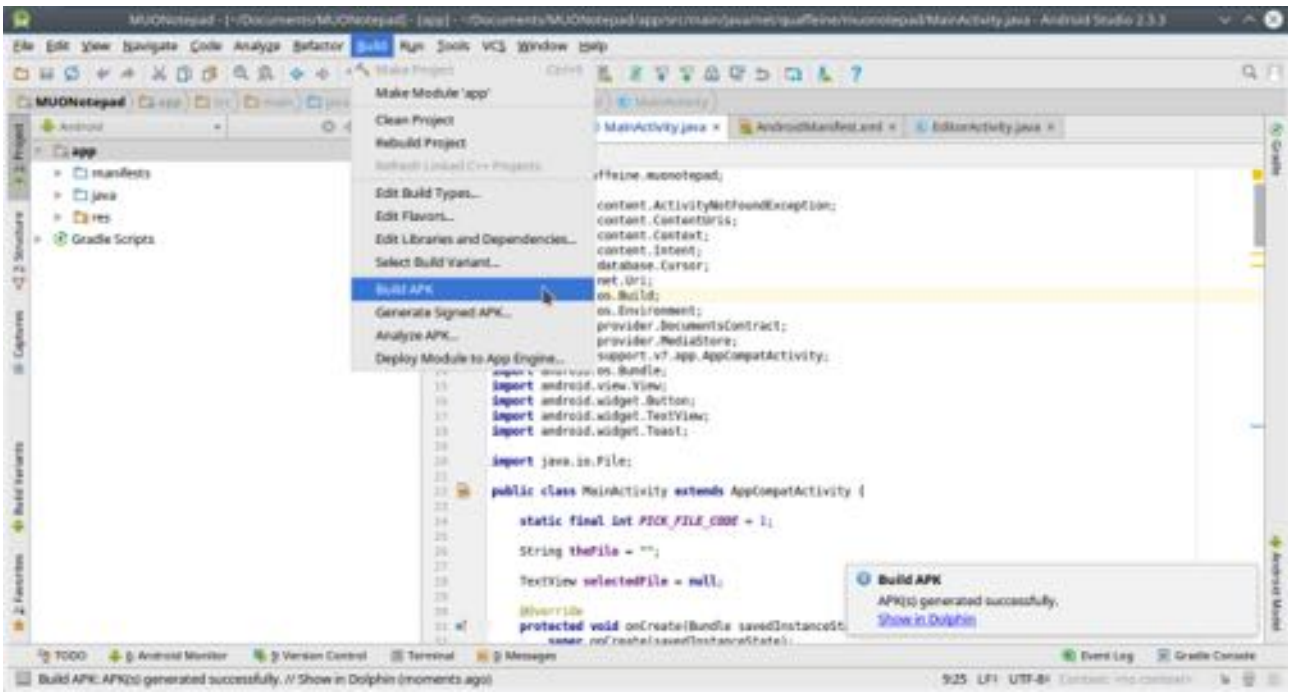
Head over to the **Build** menu, and select the **App (save .apk to my computer)** item. The app will start to build (evidenced by a progress bar), and once it completes, you'll get a dialog save the APK file. Now you can copy and send it to your heart's content.



In order to install the app, users will need to allow third-party software installations in the device's settings as described here.

Building an APK in Android Studio

Building an Android package is just as easy in Android Studio. Under the **Build** menu, select **Build APK**. Once the build is complete, a notification message will give you a link to the folder on your computer containing the app.



Google Play Distribution

Getting set up as a Google developer is a bit of a process. While you should by all means consider it once you have some experience under your belt, it's not something you need to tackle right away.

First off, it has a $25 registration fee. It also has a number of technical details that are somewhat difficult to change at a later time. For example, you'll need to generate a cryptographic key to sign your apps, and if you ever lose it, you won't be able to update the app.

But at a high level, there are three major processes you'll need to do in order to get your app into the Play Store:

1. **Register as a developer:** You can set up your developer profile (based off a Google account) on this page. The wizard walks you through a fairly straightforward registration process, which includes the aforementioned $25 fee.

2. **Prepare the app for the store:** The emulator versions of the app you've been testing are also **debugging** versions. This means they have a lot of extra code related to troubleshooting and logging that's not necessary, and they may even represent a privacy concern. Before publishing to the Store, you'll need to produce a **release version** by following these steps. This includes signing your app with the crypto-key we mentioned earlier.

3. **Set up your infrastructure:** You'll also need to set up the Store page for your app. Google provides a list of advice for setting up a listing that will get you installs (and sales!). Your infrastructure may also include servers with which your app will sync.

4. **Lastly,** if you want to get paid, you'll need a payment profile. This is one of those **once-and-done** details, so make sure you know how everything will fit together before moving forward.

Summary and Lessons Learned

We've come to the end of the guide. Hopefully this has piqued your interest in Android development and given you some motivation to take your idea and actually develop it. But before you put your head down and start building, let's look back at some of the key lessons we learned in the above sections.

- We looked at **two paths** to make your app: point-and-click builders, and coding from scratch in Java. The first has a lower learning curve and offers a fair (yet still limited) assortment of functionality. The second allows you to build just about anything you can think of and offers benefits beyond Android development, but it takes longer to learn.
- While they each have their pros and cons, **you can use both paths!** The point-and-click environments offer a quick turnaround to prototype your app, while the second allows you to re-build it for long-term improvement.
- While it's tempting to jump right into working on the app itself, you'll be very glad later if you take some time to **design your app**, including sketches of the interface and/or informal documentation on its functions. This can also help you determine if one or both of the methods above are good options.
- An easy way to start developing is to lay out user interface elements, then "wire them up" by programming their functionality. While experienced developers can start coding "background" components, for newbies, it helps to be able to visualize everything.
- When diving into code, don't be afraid to search the web for answers. Running a Google search with a couple of keywords and "code example" at the end will get you some good results.
- As you're building, test your work a little bit at a time. Otherwise it will be very difficult to determine which of the past two hours' actions broke your app.

With these in mind, get in there and start making your app-development dreams come true. And if you do decide to get your hands dirty, let us know how it goes in the comment section (we love links to screenshots, by the way). Happy building!