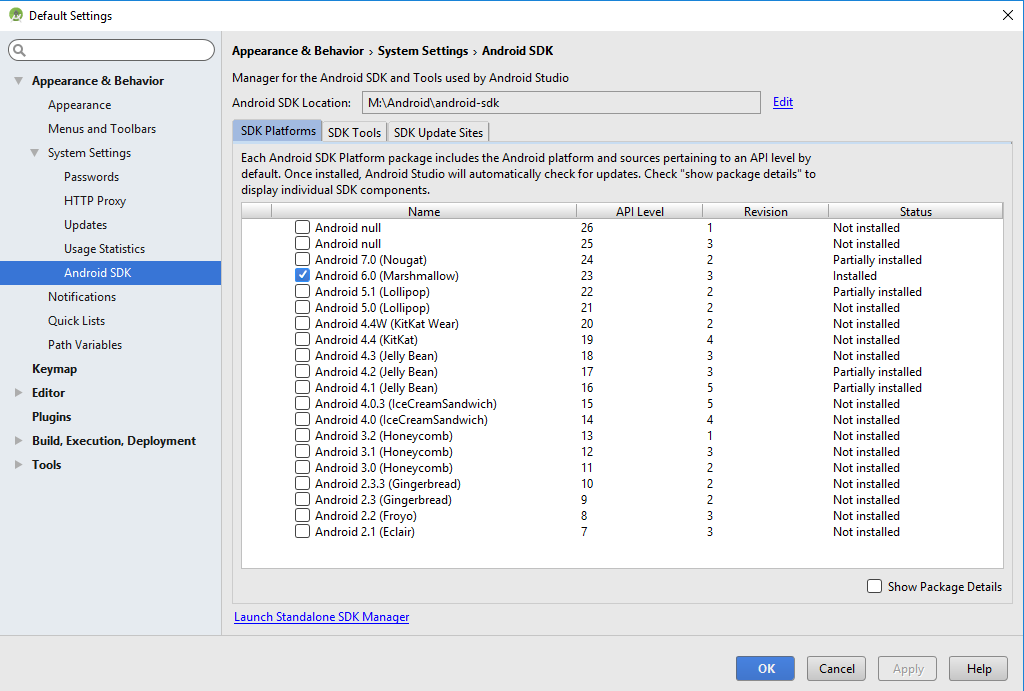
# Building Mobile Apps

In this class, you will learn how to get started with Android development by creating a simple calculator app. To begin, you will need to install Android Studio as well as the Android SDK. You can find screenshots to walk you through the installation process in the Android Environment Setup file.

## Getting Started

### SDK

For this class, we will use building for Android 6.0 so we need to get the appropriate SDK. Once you have Android Studio installed, navigate to Tools -> Android -> SDK Manager. Under the SDK Platforms tab, be sure to mark the check box for Android 6.0 (Marshmallow) and click OK. If the SDK version was not already installed, it will download the SDK and emulator images need to run against this version of Andorid.



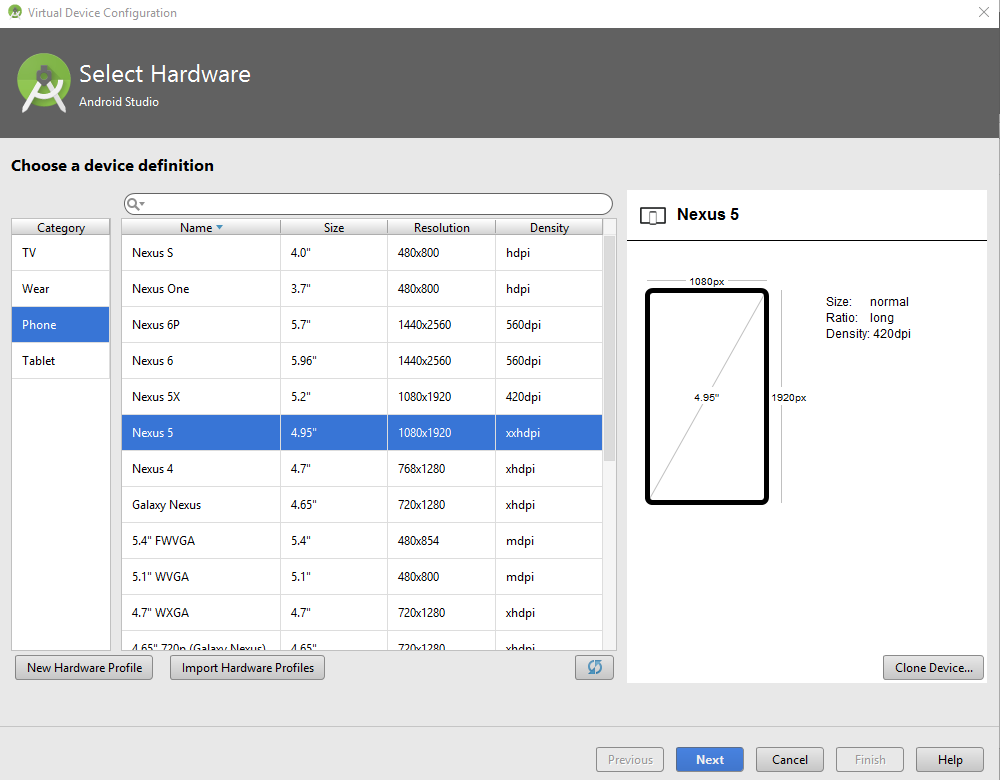
Having multiple versions installed would allow us to access features in newer version of Android test against different version of Android. Since support for certain features can change with the version number, it can be important to have multiple version install when creating production apps. For this class, we will be creating a simple app that we won’t be publishing so it’s safe to just work with a single version.

### Emulator

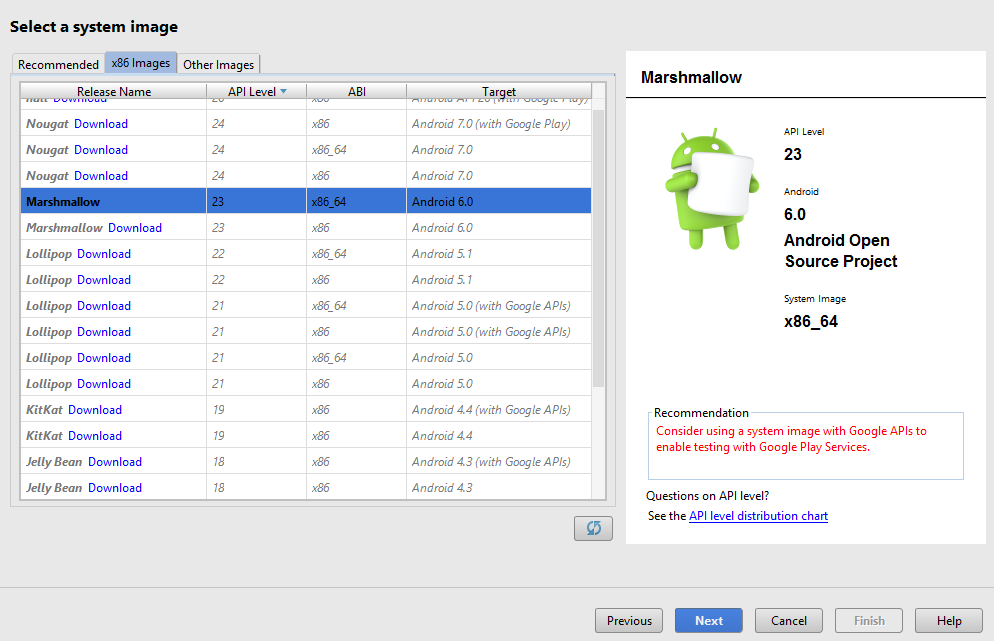
In order for us to be able to test our app, we have to options: use a physical device or test on an emulator. The emulator allows us to do all the testing on our computer and it’s easy to be able to try out our app and different Android versions and device sizes. If we wanted to only test on physical devices but still wanted to test with all the different sizes and version, we would have to spend thousands of dollars to buy all the devices.

In our class, we are going to use an emulator that is similar to the Nexus 5 device. In Android Studio navigate to Tools -> Android -> AVD Manager. Here you will find a list of all the emulators that are installed on your machine. The following steps will walk you through creating the virtual device that we will using in this class.

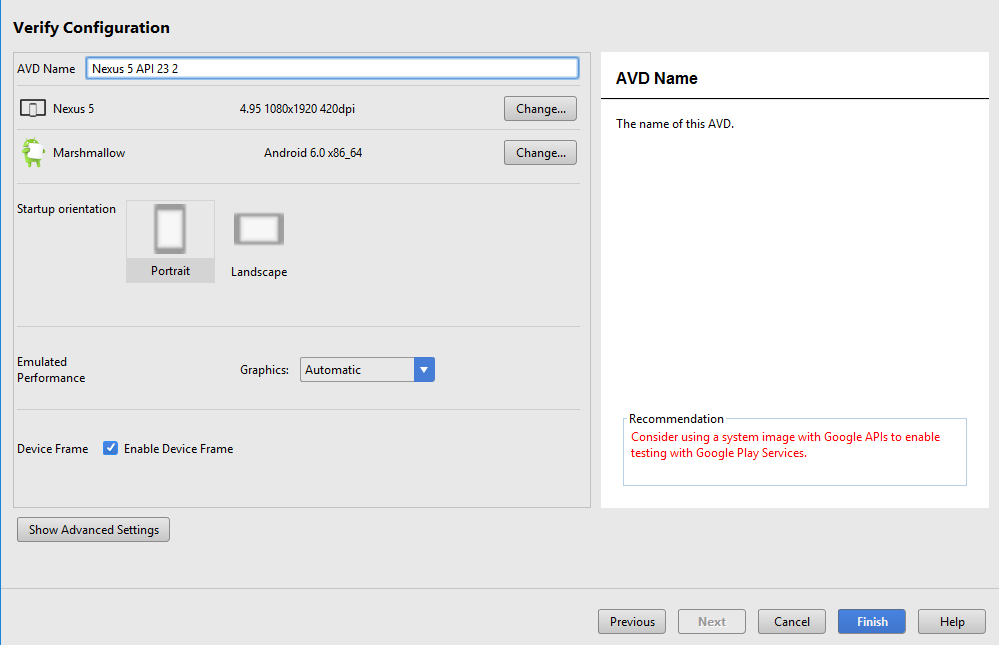
1. Click the Create Virtual Device button at the bottom right.
2. Select Phone for the Category and Nexus 5 for the Name of the device. Click Next. This screen allows us to select the size of the device that we would like to use based on real devices.



1. Click on the x86 Images tab and select the Marshmallow x86\_64 image. Click next. This screen allows you to select the version of Android that will be running on your emulator.



1. On the next screen, we will keep all the default. Click Finish.



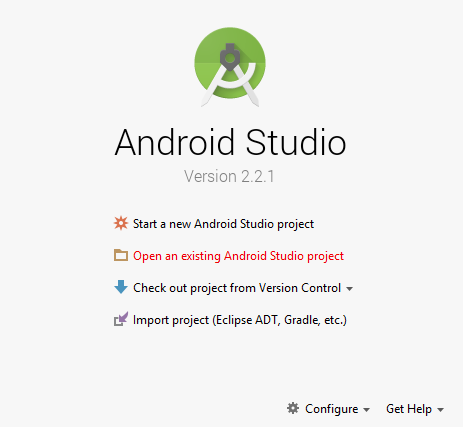
If you followed the steps above, you will now have an emulator that you can run your app on.

### Starting Project

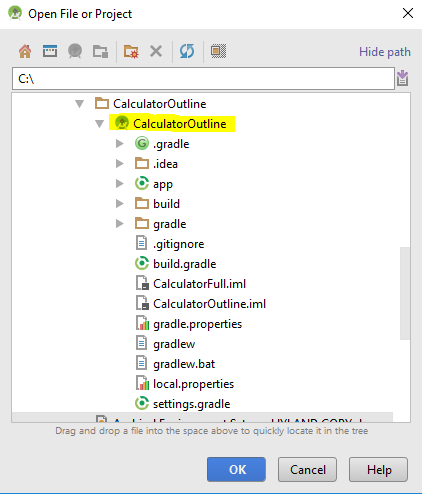
In this class, we will be using a partially completed project to help us get started. It will allow us to skip some of the steps to setting up the project and allow us to get into some of the more important topics and hopefully get a working calculator by the end of the two days.

You instructor should have given you a link containing the starting project. Navigate out to that link and download the zip file. You will have to extract the zip file somewhere on your machine.

Luckily for us, the project is all set up and ready to go so we just need to up it up in Android Studio. Launch Android Studio and on the starting window, click the button labeled “Open an existing Android Studio project”.



Next, browse out to the location where you unzipped the downloaded file and you should see the project name CalculatorOutline with the Android Studio icon next to it. Select this folder and click OK to open the project.

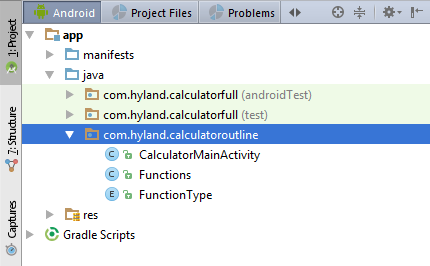


## Navigating Our Project

Once we open up our project, we will get a mostly blank view on the window. This doesn’t mean that the project is empty, it just means that we aren’t looking at anything inside of the project yet. To see the files contained in our project, we will click on the Project tab on the left hand side.



After clicking the tab, a panel will open up and we will be given a view of the files in our project. At the top of the panel, we are given multiple options for how we would like our files to be organized and what information we would like to see but for this class, we will stick with the Android view. In this view, navigate the folder structure to app/java/com.hyland.calculatoroutline.



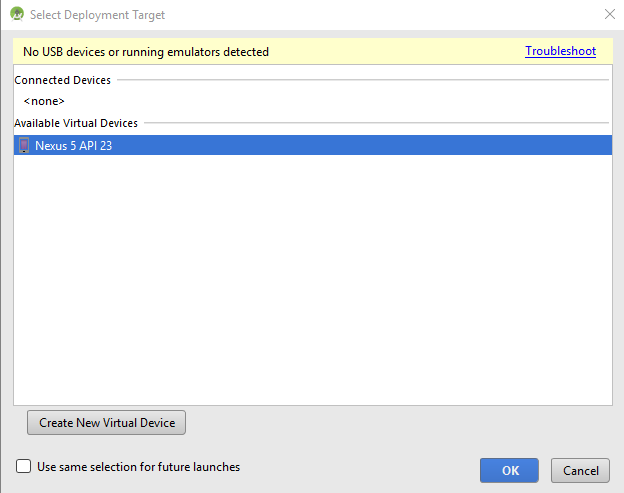
Here, we will see the three source code files that we will use for our app: CalculatorMainActivity, Functions, and FunctionType. Double click on any of these files and you will see the code in the file.

## Running Our App

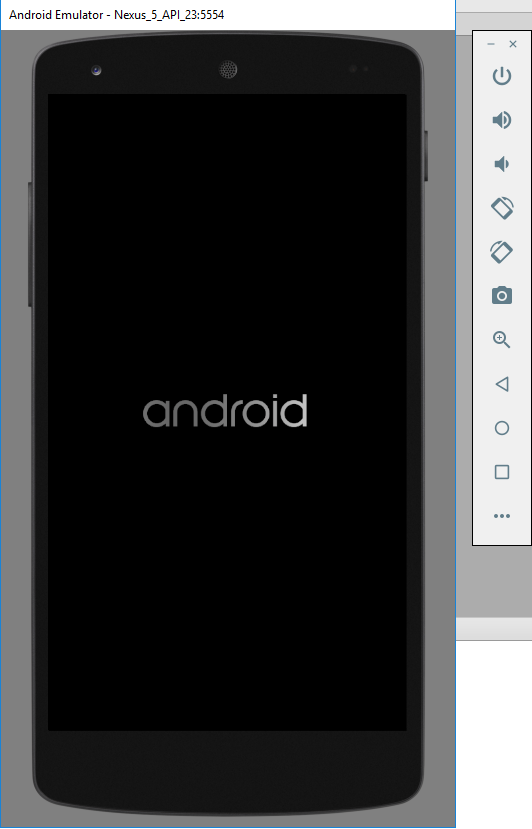
One of the things that we will do throughout the class is running our app so that we can test the changes that we have made so far. Let’s start out by running the base project that we have been given. At the top of Android Studio, click the Run icon that looks like a green play button.



Then you will be prompted to select the device or emulator that you want to run the app on. Here, we will select the Nexus 5 emulator that we created earlier. Once the emulator is selected, click the OK button to launch the emulator and build the app.



The emulator should launch and look something like the following as it loads up:



Give it a minute or two in order for the emulator to load. Once it’s loaded, the app will be installed on the emulator and launched. The app will look as follows:



At thing point, the app does not do anything. You can try clicking all the buttons but nothing happens. That’s because we need to write the code to tell the app what to do when each of the buttons are clicked.

## Exploring the Starting Code

Before we begin writing any of our own code, let’s take a look at what is already in the project.

### Resources

First, let’s take a look at how we got our calculator to look the way it does. In the Project explorer, navigate to the app/res folder. Here, you will see multiple folders and you can learn about each folder that goes in the res folder [here](https://developer.android.com/guide/topics/resources/available-resources.html).

For our project, the layout folder is what we are interested in. In this folder, we create layout .xml file that define how entire screens or parts of our screen look like. If you open up the activity\_calculator\_main.xml file, you will see a preview of what this layout file is defining the screen to look like. It should look familiar since it’s the same layout file that we used in our app and that you saw when we ran our app in the emulator.

Android studio makes it easier to work with these layout files by giving you a preview of the layout and allowing you to drag and drop other controls from the left pane into the preview to change the layout. We won’t be modifying the layout for this project but after the class, you can create your own layout file and try out the different controls to see what each one looks like.

The preview is just that, a preview. The activity\_calculator\_main.xml file is actually a text file that follows the xml syntax. But where is all the text?!?! At the bottom left of the file editor, you will see tabs for Design and Text. Select the Text tab and you will be shown the xml code that generates the preview you saw earlier. Once you get more comfortable with creating these layout files, it is often easier to just edit the text directly rather than using the drag and drop functionality of the preview page.

### Code

Now, let’s take a look at the source code files that we will be modifying to get this calculator app to work. We have three files so we will take a look at each.

#### CalculatorMainActivity

The CalculatorMainActivity class defines the first and only screen in our app. This class will be responsible for doing the computations for the calculator, handle what happens when the user clicks a button, and handles what is displayed on the screen.

Starting at the top, we see that we have defined 7 variables inside of the class. We will talk more about these when we use them as we build out the app.

Next, we have the onCreate method. This method gets called when the screen is created an allows us to do any setup work that needs to take place before the user is allowed to interact with the screen. In here, we have a call to the setContentView method.



This method tells the screen that we will be using the activity\_calculator\_main.xml layout file to define what the screen will look like. Any time that we want to refer to a layout file, we use the *R.layout.* prefix to indicate that we want to refer to a layout file. The same can be done for other items in the res folder by replacing *layout* with the type of resource that we are trying to access. For example, later on in the class, we will need to refer to identifiers that are defined without our layout files so we will use the prefix *R.id.*

Continuing down the onCreate method, we have an if statement checking to see if the savedInstanceState is null. We will be filling in this if statement later but know that this is needed so that we can handle recreating this activing if it is ever destroy, such as in the case of device rotation.

Further down in the method, we have a few calls to setup methods that are defined later in this class. The purpose of these methods will be to perform the setup steps on the buttons so that they are ready for the user to interact with the calculator. If you look at the methods now, they are all blank so we will need to fill those in.

After the onCreate method, we have the onSaveInstanceState method. This method works hand in hand with the if statement above to save the state of the activity when it gets destroyed. We will be filling this in later in the class.

Next, we have the getDisplayValue and setDisplayValue methods that we have defined. These methods are responsible for getting and setting the value that is shown in the display portion of the calculator. We will use these as helper methods whenever we want to get the current displayed value or we want to change it. The way we do this is by getting a reference to the TextView controls that displays the value on the screen.



In the line above, we use the findViewById method to get us a reference of the control that represents the id *R.id.text\_view\_display*. If you take a look at the layout file, you will see where this id is defined. The id gives us a way to get a reference to the control from the code. Once we have a reference to the TextView, we can call either the getText or setText methods to get or set the text in the control.

Continuing down the file, we have the performFunction(String currentNumber) method. We will use this method to perform the calculations based on the currently selected operator (add, subtract, divide, or multiply). Once the calculations are done, it will return the result of the calculation.

The next method, concatDisplayView, will be used to update the value that is displayed in the calculator when one of the number buttons are pressed.

Finally, the last method in our starter project, onClick, will be used to handle the event of a use clicking on a button. We will see that this is one way that we can handle button clicks but we will also handle them using a different code structure later on in the class.

#### Functions

The purpose of the Functions class is to define functions that will take in two numbers and perform an operator on them. We separate out these methods into a separate class so that our first class does not get crowded and the methods we are writing in this class can be reused for other projects so its easier to reuse them if they are in a separate file.

#### FunctionType

The FunctionType.java file contains a single enum that will help us define the type of operator we are trying to perform on the numbers.

## Handling Button Clicks

Now that we have some understanding about what we are working with, let’s get started with writing some code. Our first task will be to get the number buttons to actually do something. Currently, the app doesn’t do anything when you click on any of the buttons so we need to fix that.

To start off, we are going to do something simple and just display the last number that was pressed. In an actual calculator, we would expect the number we pressed to be added onto the number displays so that we can enter in multi-digit number but for this first step, we will keep it simple and then build up on it.

Let start by going into the setupNumbers() method. Just like we got the reference to the TextView control in the getDisplayValue() method, we need to get the reference to our number buttons here. Let’s do so by using the findViewById method and the id that we defined for each of our number buttons in the layout file:



This will look in our Activity for a Button with an id of *button\_0*. Once we have this button, we need to tell it that we want to listen to the click event so we know when a user presses the button.



This tells the numberButton to set the current activity as the ClickListener for the button. That means that whenever button is clicked, it will notify *this* object by calling into its onClick(View view) method.

We don’t have just one number button for we also need to do the same for the rest of the buttons. For each one, we need to get a reference to the button using findViewById and then call setClickListener on each button. We will set the click listener as *this* for each of the buttons so that we can have a single method that handles each button click. Here is an example of what we do to assign the click listener to the number 1 button but you should follow the same pattern for buttons 2 through 9:



Now that we will be notified when a button is clicked, we actually need to do something when the button is clicked. We will react to the notification in the onClick method. The view parameter that we get with this method is the view (Button in our case) that was click. Our first step will be to figure out which button was click. Let’s start by taking the view that we are given, casting it to a Button since we know it will be a button, get the text of the button, and assign that value to a String variable called clickedNumber.



Next, we will simply make a call to the setDisplayValue method so that the clicked number gets set as the display value.



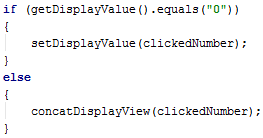
Now it’s time to run our app to see if our code changes work! Click the Run button at the top and wait for the emulator to load up and the new version of our app to be installed. Give the number buttons a try and now your display value should update each time you click one of the number buttons!

## Displaying Multi-Digit Numbers

Now that we have the numbers displaying, let’s add onto our onClick listener method so that we can display multi-digit numbers. This still won’t cover everything we will need to check for when we click on a number button but it will get a closer.

When we are building up the multi-digit number, it might seem as simple as just tacking on the number clicked onto the end of what is already being displayed. However, that could end up causing some trouble since we could end up with a number like “01234” when it really should have been “1234”. Because of this, we must to see if there is a leading zero and remove it before we display the clicked number. To do this check, we will start off we an if statement to see if the current displayed value is a zero. If the displayed value is a zero, we will simply display the number that we clicked. Otherwise, we will need to add on the clicked number to the number that is being displayed. We will leave the concatenation to be done in our concatDisplayView method. For now, try writing the if statement following the description above.

Once you have you solution, compare it to the following:



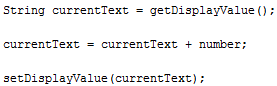
We will use this if statement to replace the setDisplayValue call that we were doing before to always replace the displayed value.

We will leave the onClick method be for now but we did say that we were going to be using the concatDisplayView method so now would be a good time to actually make it do something. The goal with this method is to add on the new number to the end of the number that is already being displayed.

Try this out yourself. Here are a few hints that might help you along:

* We have methods to both get and set the current displayed value
* You are getting the clicked number as a parameter
* String concatenation(joining) can be do using the + operator on two strings

Once you have a solution, compare it to the following:



Try this out and you should now be able to build up a multi-digit number by clicking multiple number buttons!

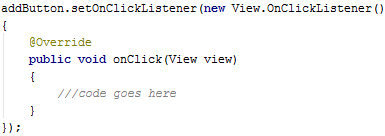
## Using the Operators

Our next step will be to use the operators and be able to provide both numbers that will be operated on. Up until this point, we have only been working with a single number but we know that it takes two to add so how do we let our user put in the second number?

The answer is that we have to remember the number that has already been typed in and then allow the use to start typing in the second number after clicking on an operator. Let do this in baby steps and first just worry about adding a click listener to our operator buttons. Remember that we have separate setup methods for each operator so we just start off with the setupAdd method. In here, we will be dealing with the add button so let’s first get a reference to it.



Next, we need to add an OnClickListener to this button so that we can be notified when it gets clicked. We can do what we did before and just say that *this* activity will listen for the notification but that can get kind of messy if we have our onClick method listening for the notifications from both the number buttons as well as the operator buttons. For the operators, let go about it another way by creating something called an anonymous class. What that will allow us to do is to define an object that can serve as the OnClickListener for the button and keep it separate from the OnClickListeners of all the other buttons. The following is the syntax for anonymous classes:



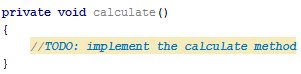
Now when the add button is clicked, the onClick method in this anonymous class will be called rather than the onClick method that we were using for our number buttons. Cool, right?!

To fill in our onClick method, we first need to decide all the things that need to happen when we click a button. Here is a list that we can start with:

* First, we need to calculate the total of everything that has been entered up until that point. This is something that will need to be done for all of our operators so we will create a calculate() method that can just be called any time the total needs to be calculated. For now, we will create the method and leave it to be implemented in a later section of the class.
* Then, we need to make sure that we aren’t adding more digits to the number that is currently displayed and instead start a new number. This is needed because when we click the operator, we want the start entering the second operand next.
* The final thing we need to do it remember what operator was pressed so we know when performing the calculation.

Let’s work through that list one by one by filling in the onClick method for the add button. Once we have the add button complete, we will be able to do something very similar for the other operator buttons.

Let’s first create our calculate method. We won’t implement the method for now but we will put in a comment to remind ourselves to fill it in later. It doesn’t matter much where you place the calculate method but make sure that it have void listed as the return type and does not take any parameters. It should look like the following:

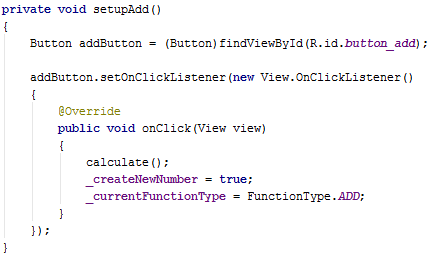


Now we can return to the onClick method for the add button and make a call to the calculate method.

Next, we want to make sure we start typing in a new number rather than concatenating with the number being displayed. To do this, we just need to remember that we want to create a new number for the next time that we click on of the number buttons. This is easy because we can use a class variable to do this. In fact, we already have the \_createNewNumber variable defined so we just have to set the value. In the onClick method, we will set its value to true, indicating that the next time a number button is pressed we will be starting a new number.

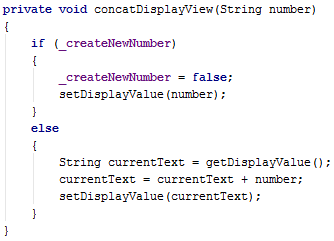
Finally, we have to keep track of the operator that we pressed so that information is available to use the next time we perform a calculation. Again, we already have a variable available to us numbed \_currentFunctionType so we can just assign it a value. The interesting thing about this variable is that its type is an enum named FunctionType. An enum is a type that is defined to store a set number of values. In our case, the values are NONE, ADD< SUBTRACT, MULTIPLY, and DIVIDE. IF you would like to see how this enum gets defined, take a look at the FunctionType.java file. For our add button, we will assign the value FunctionType.ADD.

With these changes, our setupAdd method is not complete and should look like what is below. Take what you did for the setupAdd method and fill in the implementations for the setup methods of the other three operators.

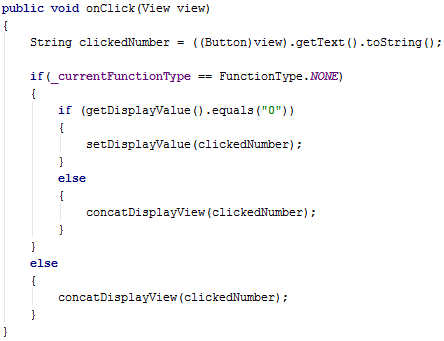


Before we go ahead and test the operator buttons, we still need to use the \_createNewNumber variable to tell us when to create a new number. A good place to do this check would be in the concatDisplayView since that is where we are updating the display view.

We already have some code in here so we can just add onto what we already have. We said that we want to create a new number if \_createNewNumber is true and we will continue adding onto the current number if it is false. This sounds like we need an if statement. Let’s create an if statement that checks for the value of the \_createNewNumber variable. IF the value is true, we will set the value to be false and set the displayed number to be the number that was passed in. If \_createNewNumber is false, we will continue to do what we already had in the method. Try writing the if statement to match the description above and then compare your code with what is below.



The final change that we need to make in this section is to verify that we always check for creating a new number when a number button is clicked. In the onClick method we should keep our current code in the case that the \_currentFunctionType is FunctionType.NONE but if it is one of the other function types, we need to call the concatDisplayViow method so that the check for creating a new method takes place. You can use an if statement for this. Try it out and then compare your modified code with what’s below.



Now we are ready to test the app to check our progress. At this point, we should be able to type out a multi-digit number, click an operator button, and the next time a number button is clicked, the displayed number will be cleared to show the new multi-digit number. Give this a try to make sure that it continues to work with all of the operator buttons and not just the add button.

Of course, none of the operators actually do any calculations yet but that’s because we haven’t implemented the calculate method yet. We’ll take care of that next.

## Making the Calculator Calculate

Now it’s time for the fun part!! Let’s make our calculator actually calculate!!

At the end of the last section, we were able to get our calculator to clear out and start new number as well as remember the operator that was pressed. Now we will used the information about the operator that was pressed in order to perform the calculations. In this section, we will need to modify two methods: calculate and performFunction.

### calculate Method

If you remember from earlier in the class when we were implementing the methods to setup the operator buttons, we called the calculate method with the expectation that we would implement it later so let’s do that one first. Our calculate method will be responsible for keeping track of the previous total and executing the operator that had been last clicked. Since our calculator will start a new number after clicking on an operator, we will no longer know what the left hand operand is. By having the calculate method save off the previous total right after performing a calculation, the left operand is saved off even after it’s cleared from the display.

Our first step in the calculate method will be to get our right operand, which conveniently is the number that is being displayed. We will save this off in a variable called rightOperand.

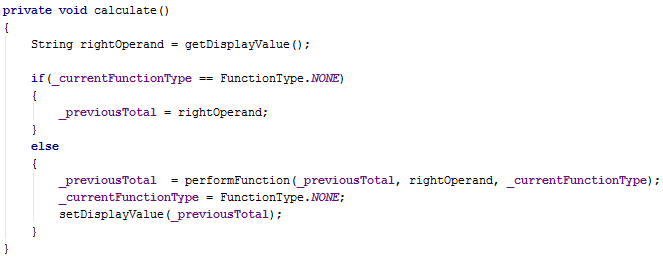


Next, we have two different cases that we care about. The first is that we are just starting off our equation and have just clicked our first operator button. The second case is that we have already selected an operator and actually need to do a calculation. To handle both of these cases, we will use an if statement. For our first case, we will check to see if the \_currentFunctionType has a value of FunctionType.NONE. Otherwise, we will have an operator to perform a calculation with and that will fall into our second case.

Now, for our first case, since we do not have a function type selected, we do not need to perform any calculation but we do need to save off the displayed value so that it can be used the next time we do a calculation. To do this, we will save the value of our rightOperand variable into the \_previousTotal class field. We will use this \_previousTotal as the leftOperand the next time we perform a calculation.

Then, in our else statement to handle our second case, we will need to perform the selected operator, reset the \_currentFunctionType, and set the display value to be the new total. First, to perform the calculation, we will be off loading that onto the performFunction method to keep our calculate method from getting too long. Our performFunction method requires us to provide three parameters: leftOperand, rightOperand, and functionType. As we mentioned earlier, the value stored in \_previousTotal will serve as our leftOperandValue. We already have our right operand stored in a variable. The functionType that we pass in will be the one stored in the \_currentFunctionType variable. We will then save off the output of the performFunction method as the \_previousTotal so that it can be used as the left operand for the next calculation. Once that calculation is performed, we can reset the \_currentFunctionType to FunctionType.NONE in preparation of the next operator. Finally, we need to show the total to the user so we will set the display value to be the \_previousTotal.

This should wrap up the implementation of the calculate method. Check you code with what’s provided below:



### performFunction Method

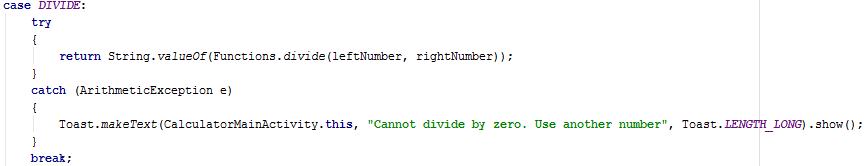
Now, we need to implement our performFunction method since it’s used in our calculate method. The purpose of this method is to take in two operands and an operator and perform the calculation. This method will also help us do some error handling since we can run into some trouble if we try dividing by zero. Also, we have some code in the Functions.java file that we will use. The problem is that the methods in the Functions class require us to provide the values as longs but all we have been dealing with so far is Strings. We will have to do some conversions and will handle that in the performFunctions method.

First, we will handle the problem of converting the String operands to longs so that we can use the methods in the Functions class. Let’s create a variable of type long and name it ‘leftNumber. To get the value to assign to this variable, we will need to use the Long.valueOf(String value) method. This method will take a String and convert it to the long equivalent of the number in the String. Call the Long.valueOf method with leftOperand as the parameter and assign the output of this method to the leftNumber variable. Do the same for the right operand and save the value into a variable of type long with a name of rightNumber.

Next, we will have to determine the operator that we are dealing with so that we know which method in the Functions class to call. To do this, we will use a switch statement and switch against the functionType that is passed in. We have four different operators that we can perform so we will create a case in the switch statement for each corresponding value in the FunctionType enum. In each case, we need to return the result of the operation performed on the two operands.

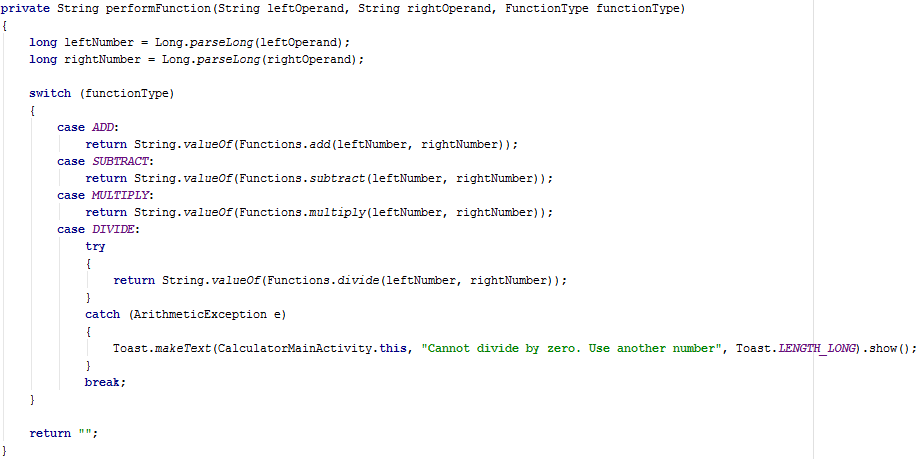
Let’s take the add operator as an example. We are going to called the Functions.add method and provide it with the left and right numbers. The problem with this is that the return value of the add method is a long but we need to return a String in the performFunction method. We can convert the output back to a String by passing it into the String.valueOf method. The output of the String.valueOf method is what we would return for the performFunction method. Repeat this code for the subtract and multiply operators but hold off on divide since we have to do a little extra work for it.

In order to call the Functions.divide method, we need to be prepared in case the user is trying to divide by a zero. If this happens, The Functions.divide method will throw an ArithmeticException which can cause our app to crash if it is not handled properly. Exceptions are a way in a program to state that something happened that is not a valid state for the application. The example we have in our app is dividing by 0 since that is not a valid operation. In order to properly handle the Exception, we have to tell the program what to do in the case that the ArithmeticException is thrown. We do this by wrapping the method call in a try-catch statement. The catch part of the statement is where we tell the app what to do in case of an Exception being thrown. Copy the code below for the divide case.



The Toast.makeText method is an Android method that allows us to show a toast message on the screen. The first parameter, CalculatorMainActivity.this, defines the context that the toast message is being created and displayed in. The second parameter is the String that we want to be displayed. The third parameter defines how long the toast message will be displayed before it fades out.

We should now be complete with the performFunction method and you can compare your code with what’s below:

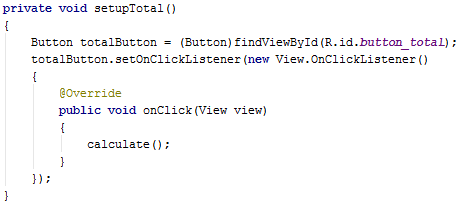


Now we get to the fun part!! Run your app in the emulator and make sure that your operators work. We still have some buttons that don’t work but you will now see the calculate performing the operators as you chain together multiple operators.

## Equals Buttons

It’s a bit annoying to have to click one of the operator buttons just to see the total even when you are at the end of your equation and don’t want to perform any more operations. That’s what the equals button is for but ours isn’t implemented yet so that’s what we will tackle next.

We have had plenty of practice listening for click events on buttons so we will leave this one to try on your own. You can use the code that you wrote in the operator buttons section of the class as reference. The id for this button is button\_total and the only thing that we need to do is call the calculate method. Write your code and then compare to what’s below:



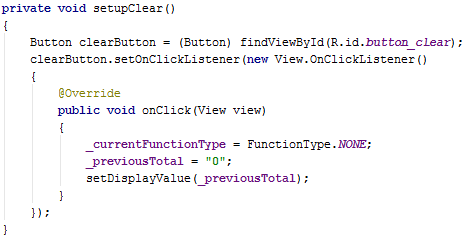
As always, test your app to make sure everything works.

## Resetting the Calculator

Another useful button that would be nice to get implemented is clear button. This will allow us to reset the calculator to its initial state so that we can start other calculations. Again, we will let you implement this on your own since it’s just another instance of handling button clicks. This time the button id is button\_clear and there are three things that need to be done when the button is clicked:

* Reset the \_currentFunctionType to FunctionType.NONE
* Set the previous total to “0”
* Set the displayed value to “0”

Write your code and then compare to what’s below:



Again, test to make sure everything works as expected.

## Handling Rotation

At this point, we have all the functionality we were planning to include in the calculator. You can add, subtract, multiply, and divide. You can even reset the calculator to its initial state by clicking the Clear button. However, there is one small thing that we are missing and it doesn’t even have anything to do with calculators.

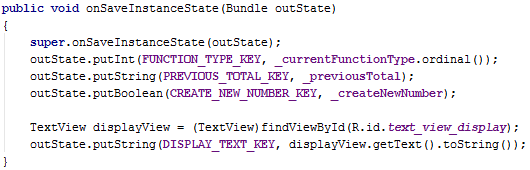
Run the emulator and do a few calculations so that you display shows something other than the initial “0” in the display. Then, click on one of the rotate buttons for the emulator (shown below).

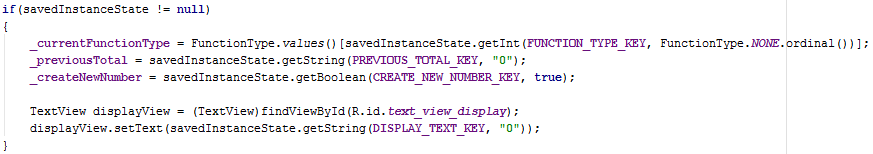


What happened?!?! You just lost the calculation that you were working on and the calculator reset to its initial state without you even clicking the clear button. This sounds like the app is broken, right? We implemented all the calculator features, why would it be broken? It turns out this resetting the state when the device is rotated is actually expected functionality. The good news is that there is something we can do about it.

When the device is rotated, the Android system will actually warn us that the Activity will be destroyed and give us an opportunity to store off any state information that we will need when the Activity is created again.

To get started, we will look at the onSaveInstanceState method. This is the method that will get called when Android is giving us the opportunity to save off our state. The outstate Bundle variable that is passed in as a parameter will get saved off and sent over to the new Activity when its created so any information that we want to save off, we have to that Bundle. We can add values to the outState as key values pairs. The four things that we would need to save are the \_currentFunctionType, \_previousTotal, \_createNewNumber, and the value being displayed. To save off this information, we can use the code below:



The other half of this is that we need to restore these values when the new Activity is created. If we remember from earlier in this class, the onCreate method gets called when the activity is being created and at that time, it’s provided with the Bundle that we modified in the onSaveInstanceState method. Scroll down to the if statement that is checking to see if the savedInstanceState is null and add in the following code that will restore the values. 

Rerun the app and try rotating after performing some calculations. This time you should notice that the calculator maintains it’s state and you can continue your work even after the device has been rotated and the Activity recreated.

## Now What?

Congratulations!!! You just created you first Android application and it’s a working calculator. This app is not as simple as it looks but you made it all the way through.

Now what can you do? Although we have created an awesome calculator app, there is always room for improvement. For example, see that decimal point button? We never implemented that and our calculator currently only works on whole numbers. Also, there is plenty you can do with the layout file to change the look of the calculator. I’m sure you can come up with other great ideas so just give it a try.

In this class, we have just scratched the surface of Android development and there are so many other things that you can do in Android apps. IF you are looking to learn more, here are a few resources that we would recommend:

* [**http://developer.android.com/training/index.html**](http://developer.android.com/training/index.html)
* [**https://www.youtube.com/user/androiddevelopers**](https://www.youtube.com/user/androiddevelopers)
* [**http://www.vogella.com/tutorials/Android/article.html**](http://www.vogella.com/tutorials/Android/article.html)
* [**http://www.mkyong.com/tutorials/android-tutorial/**](http://www.mkyong.com/tutorials/android-tutorial/)
* [**http://android-developers.blogspot.com/**](http://android-developers.blogspot.com/)
* [**https://www.youtube.com/watch?v=SUOWNXGRc6g&list=PL2F07DBCDCC01493A**](https://www.youtube.com/watch?v=SUOWNXGRc6g&list=PL2F07DBCDCC01493A)