

Android Activity Book, Beta 1.0

Learning Android the fun way!

Corey Leigh Latislaw

Preface

Mobile devices have become an indispensable part of our lives in the short time they have been around.

Perhaps the most exciting opportunities lie in the ability to make an impact on not only the local, but global scale. There's a host of emerging technology spaces that simplify our lives, change the way we work, make education more accessible, and save lives.

It's an exciting time to become an Android developer! The demand for developers is high and the supply is low. Development tools are better and more accessible to beginners. The desire for Android devices worldwide is increasing at a fast pace.

This Book is Different

Lots of books start with the basics of Android, but don't cover the skills you'll need to be an effective member of an Android team, such as source control, testing, and scaling your designs.

They teach you every API under the sun, but they don't show you how to write a view that scales to the multitude of devices. They discuss the theory, but the practice is left up to the reader to figure out for themselves.

I believe technical books should be accessible, give the whole picture, link to the latest information available, and show you what you need to know to be effective in your day to day.

How to Use This Book

This book assumes Android knowledge that is covered in my “Android Theory Book.” This book is the activity-based companion that puts the theory to practice. However, if you have some familiarity with Android, you will be fine using this as a standalone book.

The exercises in this book build on each other. You should read the book straight through from beginning to end. Each chapter depends on the work from the last.

Note: The only exception to this rule is the [Styling the Calculator](#) and [Button Interaction](#) chapters. If you prefer, you can choose to make the buttons work before styling them or vice versa.

This book can be used as a *kata*. *Katas* are typically small programs that you complete over and over again to improve your practice. They help break the protectiveness we can develop over our own code and improve the speed that we write applications.



Only through practice will you become comfortable. When you come back to this book the second, third, fourth time your muscle memory will take over. You will achieve programming flow while creating apps quickly with a high level of confidence in their correctness.



Using the book this way will help you become comfortable with tooling integration, keyboard shortcuts, letting the IDE generate code for you, and Android and testing concepts.

Conventions

I use several typesetting conventions in this book.¹

Note: Used when calling out an important note or a caveat.

When interacting with the menu system to follow  or interacting with a particular  you'll see this formatting.

If you are directed to a particular file,  the ▶ full ▶ file ▶ path will be shown. If the full path has already been used in the same section, a shorthand will be used, e.g.  path.

When referencing **ClassNames**, **functions()**, XML **attributes**, or other code items inline, a special font is used.

A box with syntax highlighting is used for longer code segments.

```
public class CalculatorActivity extends ActionBarActivity
{
    @Override
    protected void onCreate( Bundle savedInstanceState )
    {
        super.onCreate( savedInstanceState );
        setContentView( R.layout.activity_calculator );
    }
}
```

Finally, key shortcuts are shown as:  +  +  +  +  +  + ²

¹When adding context or referencing sources, I use footnotes.

²This key combination is for demonstration purposes only, do not attempt at home.

Introduction

We will build a calculator through a series of exercises.³ Each chapter is based on a specific task such as creating the display or handling user interactions. It represents a small iteration of app that adds of new functionality.

At the end of this book you will be comfortable with:

- Using Android Studio to build an application from start to finish.
- Using test-driven development from the beginning.
- Creating a fragments-based app architecture.
- Styling view elements.
- Scaling your design to many devices.
- Handling user interaction.

Although it's suggested to have some familiarity with programming, the exercises break down the steps and explain background information you may need.

Note: If you wanted to get familiar with basics of programming, check out [compilr](#), or [codingbat](#).⁴

³Code samples for the book will be posted at github.

⁴I've started a separate book that lowers the programming barrier for entry. This book is expected to release in late 2015.

Test Driven Development

This book presents a guided path to learn [test driven development \(TDD\)](#) while also learning Android development. We dive deeply into this topic in [Chapter 3](#).

Tests lead our implementation. We start by writing a failing test and then code to make the test pass. Once we have a passing test, then we are free to restructure the system (aka refactor). We continue this process until the feature is finished.

Most books assume that integrating testing from the start is too overwhelming for new readers. I think you're smarter than that. I believe that giving you the tools you need from the beginning of your journey will make it easier to go further with this career path.

You are encouraged to follow the path laid out in the book even if you feel that you won't use it in future applications. You may find you like it!

Contents

Preface	ii
How to Use This Book	iii
Introduction	v
Contents	vii
1 Hello, Calculator	1
1.1 Design	2
1.2 Solution	3
2 Hello, World	4
2.1 Project Wizard	5
2.2 Choose Form Factors	9
2.3 Select Activity Type	11
2.4 File Options	12
2.5 Run	13
2.6 First Commit	16
3 Hello, Robolectric	19
3.1 Create Test Module	21
3.2 Integrate Robolectric	27
3.3 Custom Test Runner	29
3.4 First Test	34

3.5	Run Test	41
3.6	Second Test	46
4	Building the Display	57
4.1	Create Test File	58
4.2	Start Test Fragment	62
4.3	Display View	65
4.4	Default Display	73
4.5	Show Display	75
4.6	Display Styling	77
5	Building the Buttons	82
5.1	Set Up Tests	83
5.2	Button Container	86
5.3	Add Buttons	88
5.4	Show Buttons	94
5.5	Layout the Buttons	96
6	Styling the Calculator	99
6.1	Button Styles	100
6.2	Display Style	107
6.3	Button State	108
6.4	App Icon	113
7	Button Interaction	116
7.1	Toast Test	117
7.2	Add Click Listener	119
7.3	Refactor Tests	120
7.4	Refactor Fragment	122
7.5	Finish Number Buttons	124
7.6	Configure Operator Buttons	125
7.7	Equals Button	128
8	Updating the Display	137

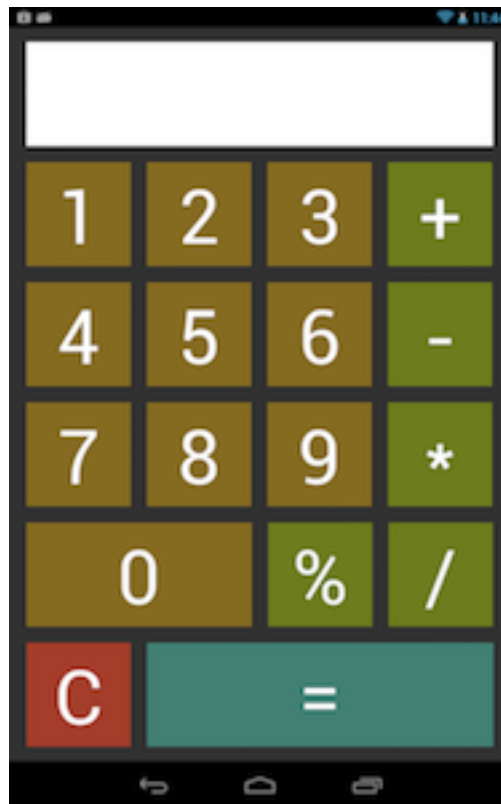
8.1	Routing Calculator Events	138
8.2	Add Application & Test	140
8.3	Setup Event Bus	142
8.4	Number Events	144
8.5	Post Number Event	149
8.6	Update Number Display	153
8.7	Update Operator Display	162
9	Calculator State	192
9.1	Adding State	193
9.2	Calculator State Fragment	195
9.3	Calculator State Fragment Test	196
9.4	Add State to the Activity	198
9.5	Migrate Activity State	200
9.6	Refactor: Base Fragment	202
10	Constructing Numbers	210
10.1	Handling Append Events	210
10.2	Append the Display	214
10.3	Set Display After Operator	218
10.4	Limit Digits	226
10.5	Storing the Operand	231
10.6	Reset the Operand	239
11	Clearing the Calculator	257
11.1	Clearing the Display	258
11.2	Clearing the Operand	261
11.3	Clearing the Operator	263
12	Handling Operators	277
12.1	Using Operator Types	278
12.2	Handling Operator Events	280
12.3	Adding Operator Types	287

12.4 Changing Operators	290
12.5 Operator First	291
13 Computing Results	293
14 Error States	294
15 Scaling the View	295
15.1 Flexible Dimensions	295
15.2 Device Rotation	297
15.3 Landscape	297
16 Appendices	298
16.1 Glossary	298
16.2 Shortcuts	298
16.3 Resources	298
17 Epilogue	299
17.1 About the Author	300

Chapter 1

Hello, Calculator

In this book, we walk through creating a calculator. We use test driven development to build out each feature slowly. We flesh out the design with colors and interactions and ensure the design scales to many form factors.



1.1 Design

There are many valid ways to break down a design. Each choice has consequences or benefits when we build out the application.

When breaking down a design, think about how you can segment it into standard view components, custom views, and fragments to best support the user experience.

There are books other resources that you can consult, but your sense of how to break down a design will evolve over time.

Exercise

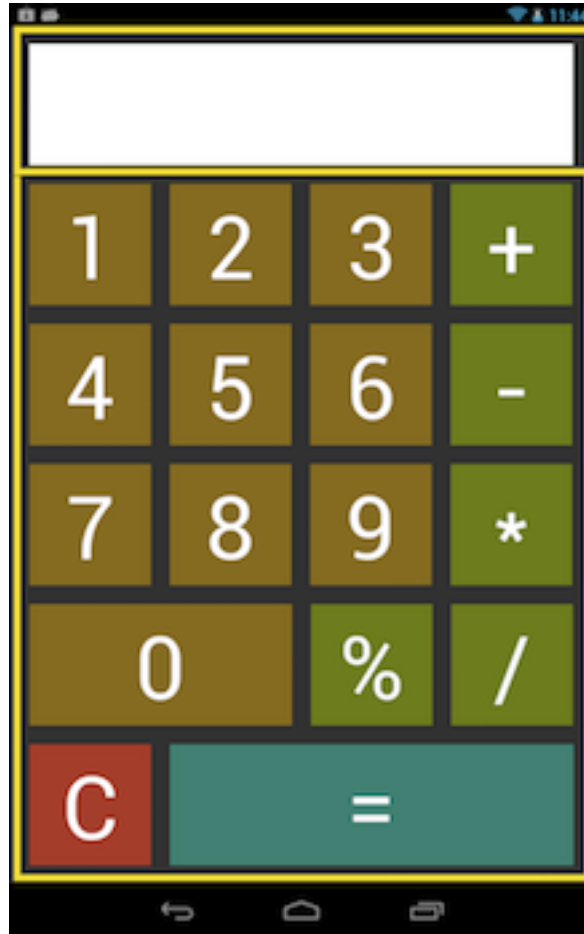
How would you break down the view? What [view elements](#) would you use?

1. Either print the design on the previous page, sketch it on a whiteboard, or draw on paper.
2. Draw boxes around the different view elements and/or add labels.

Think through at least two different ways to you could break down this view into XML components.

1.2 Solution

In this book, we'll be using two **Fragments** laid out in a **LinearLayout** with vertical orientation. We'll cover this more in depth in [Chapter 4](#) and [Chapter 5](#).



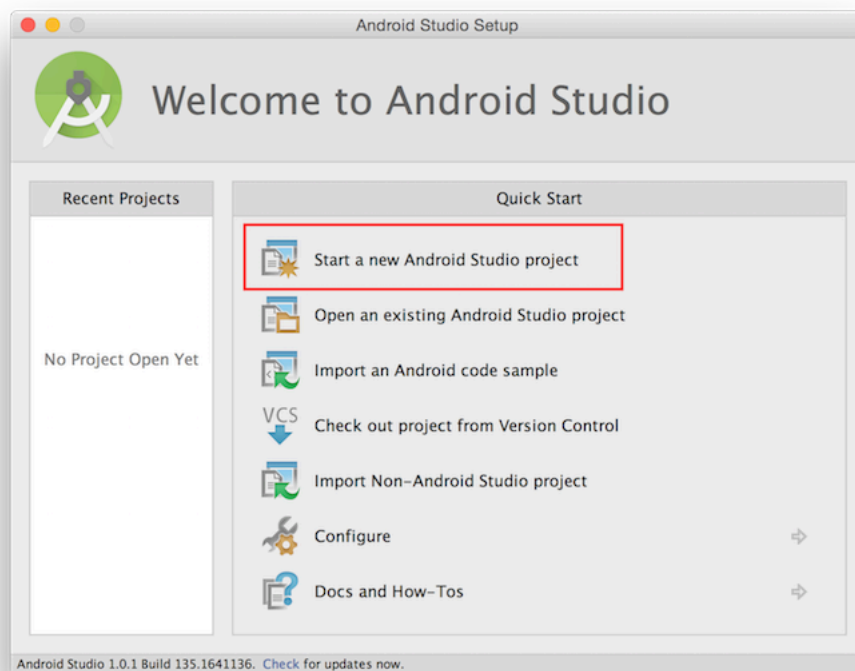
The **DisplayFragment** will handle the display. We'll use an **EditText** to display the numbers entered into the calculator as well as calculations.

The **ButtonsFragment** will contain and control the calculator buttons. We'll use a **RelativeLayout** to position many **Buttons** for numbers, operators, and clearing the display.

Chapter 2

Hello, World

We'll use Android's project templates to create our application. From the welcome screen, select **Quick Start** > **Start a new Android Studio project**. From inside the application, use **File** > **New Project...**.



2.1 Project Wizard

On the first wizard screen, fill out the following information:

- Application name: Calculator
- Company Domain: greenlivesoftware.com

The screenshot shows the 'Create New Project' dialog in Android Studio. The title bar says 'Create New Project'. The main header is 'New Project' with the Android Studio logo. Below that, it says 'Configure your new project'. There are four input fields: 'Application name' with 'Calculator', 'Company Domain' with 'greenlivesoftware.com', 'Package name' with 'com.greenlivesoftware.calculator', and 'Project location' with '/Users/colabug/AndroidStudioProjects/Calculator'. The 'Package name' field is highlighted with a red box and has a red arrow pointing to it with the text 'Automatically generated'. The 'Project location' field has a red arrow pointing to it with the text 'Changing this is optional'. At the bottom, there are four buttons: 'Cancel', 'Previous', 'Next', and 'Finish'. The 'Next' button is highlighted with a red box.

The combination of the application name and the company domain generates the **Package name: com.greenlivesoftware.calculator** (shown in gray). You can edit this name with the blue **Edit** link.

Application Name

The application name you enter will be the name of the project on disk and the title shown in the grid of app icons on your phone.

Behind the scenes, this sets the **app_name** string in `app/src/main/res/values/strings.xml`, which is configured as the application's **label** attribute in `app/src/main/AndroidManifest.xml`.

Company Domain

The company domain you enter should be a domain to which you have rights. For example, I own greenlivesoftware.com.

For future projects you plan to release, if you don't have a domain, you can buy one or create a [github](https://github.com) account. This gives you a unique `username.github.io` domain. For more information about setting this up, read the [documentation](#).

When it generates the **packageName**, the wizard reverses your domain name and adds the application name to the end.

Package Name

Behind the scenes, both the **packageName** in the `AndroidManifest.xml` and the **applicationId** in the `app/build.gradle` file are set to the same value.

Note: It's possible to decouple them, but that's not an option in the wizard. We'll leave them alone. The article [ApplicationId versus PackageName](#) discusses the differences between them.

It's preferable to choose the name you plan to ship with when you create the project. It's possible to change it later, but it's not fun (especially on a team).

The package name serves three different purposes.

1. It's a unique identifier used in the Google Play Store.
2. On a device only one application with the package name and key signature is allowed at the same time.
3. It determines where your code is stored.

You can not change an app's package name after you ship it off to the store. If you did change it, you'd need to create an entirely new application listing.

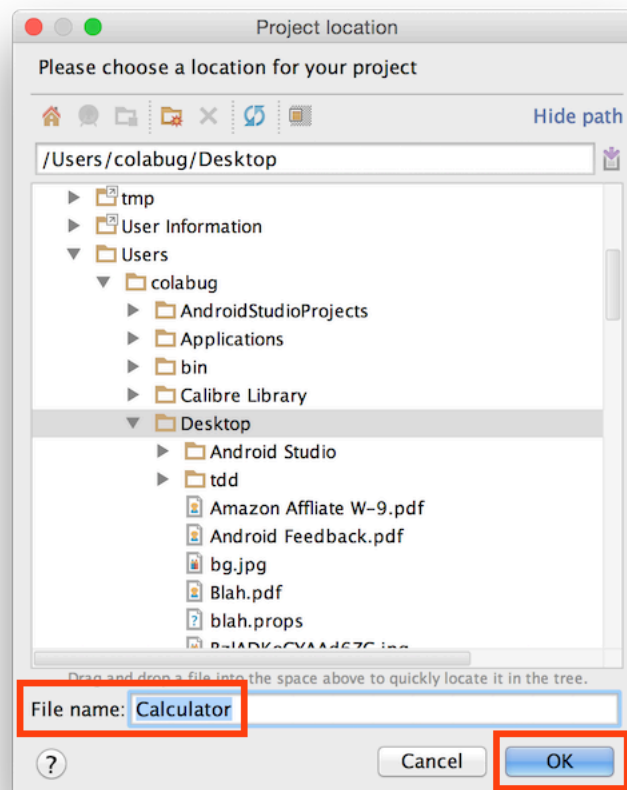
This means your current users would need to install a different application from the market instead of getting a notification that an update is available.¹ This is not only a pain for your users, but could damage your brand and make discoverability difficult.

¹For more information about shipping applications to the market, check out the [Launch Checklist](#).

Project Location

The `Project location:` is generated for you from the application name that you enter and the default project location.

You can change the name inline or open the “Project location” dialog with the `...` button to choose the new destination. Once you choose a new location it becomes the default for new projects.



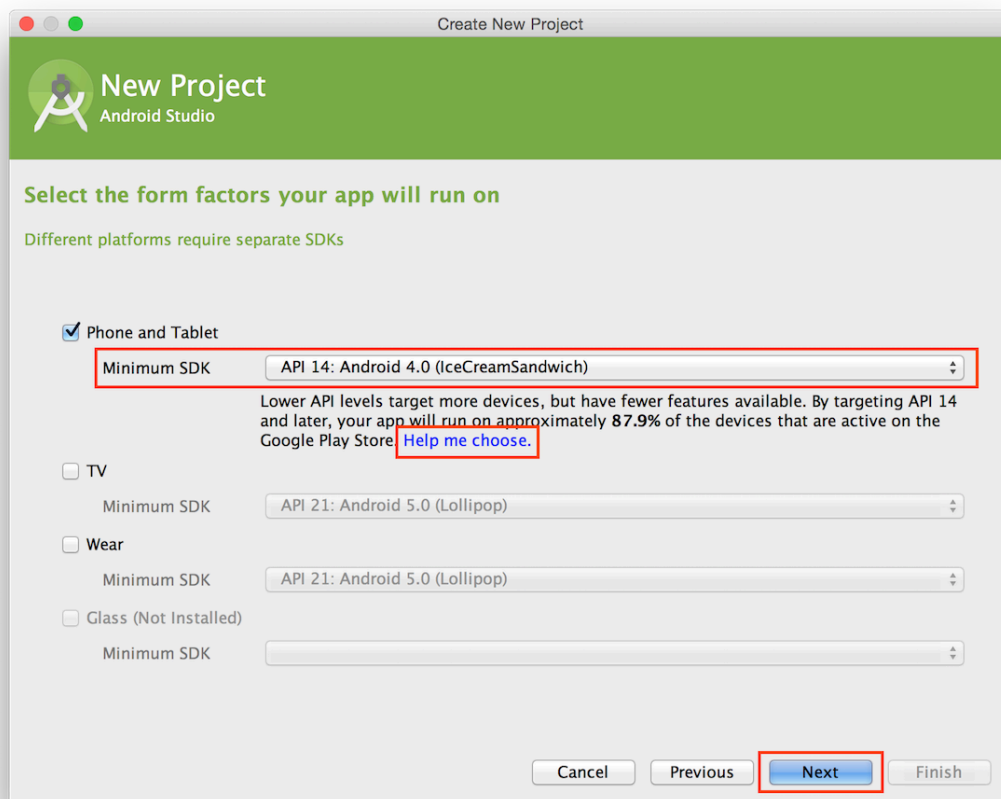
Default Mac location: `Users > [user] > AndroidStudioProjects`

Click `OK` when done.

2.2 Choose Form Factors

On the next wizard screen, we select which form factors (e.g. phone, tablet, Glass, Wear, and TV) and [API levels](#) we plan to support.

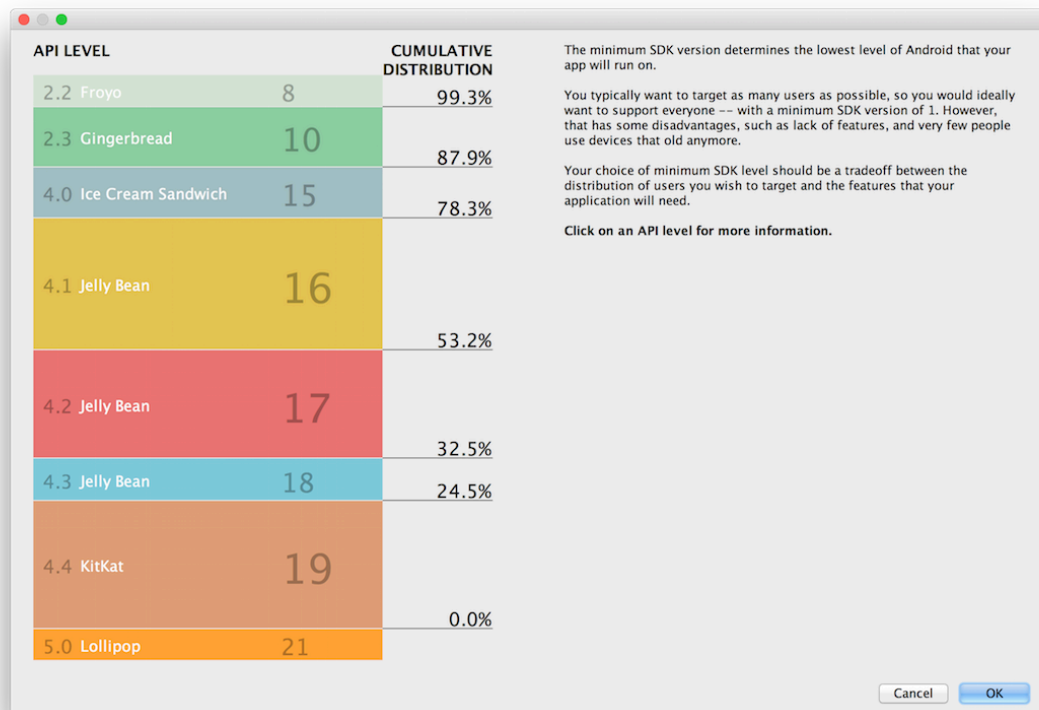
Check the `Phone and Tablet` box and select `API 14: Android 4.0 (IceCreamSandwich)`² from the `Minimum SDK` drop-down. Click `Next`.



The minimum API level you choose populates the field `minSdkVersion` in `app/build.gradle`.

²[This choice](#) offers many options for functionality and supports a broad range of devices in the market. 87.9% at the time of writing.

If you click on **Help me choose.**, you get this handy visualization of the distribution of Android OSes. You can find the latest distribution percentages on the [Android Dashboard](#).

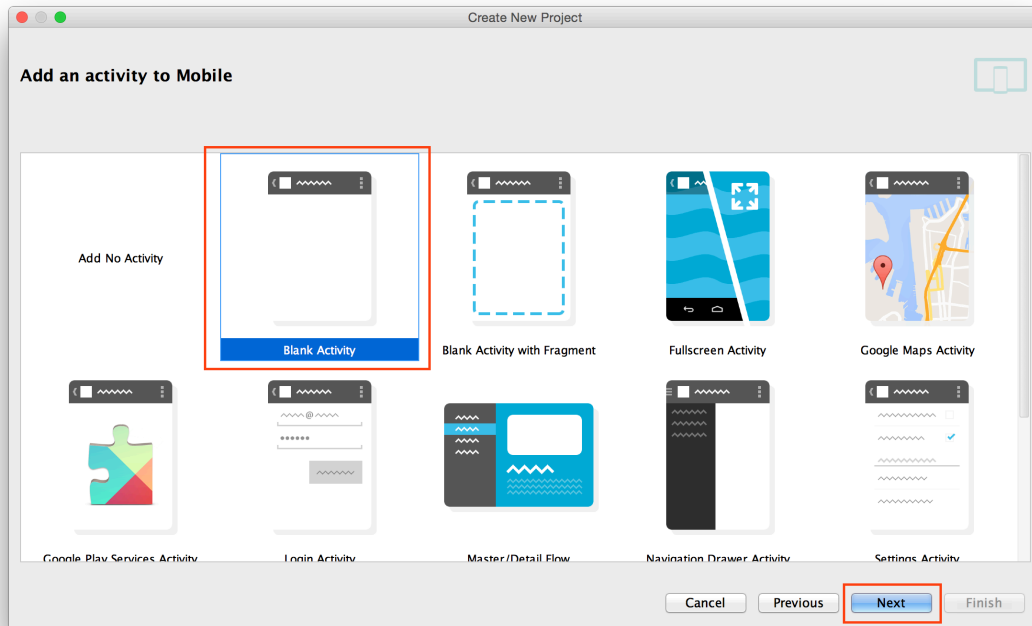


Choosing an appropriate API for your project is a subject that requires several inputs to choose the best one for your needs. The choice is influenced by your target demographic, API-specific features, devices you plan to support, current trends, and market share among other considerations.

Click **OK** to exit.

2.3 Select Activity Type

On this screen, we select the type of [activity](#). This activity occupies the complete display area of your device and serves as the entry point to the app.



Select **Blank Activity** and click **Next**.

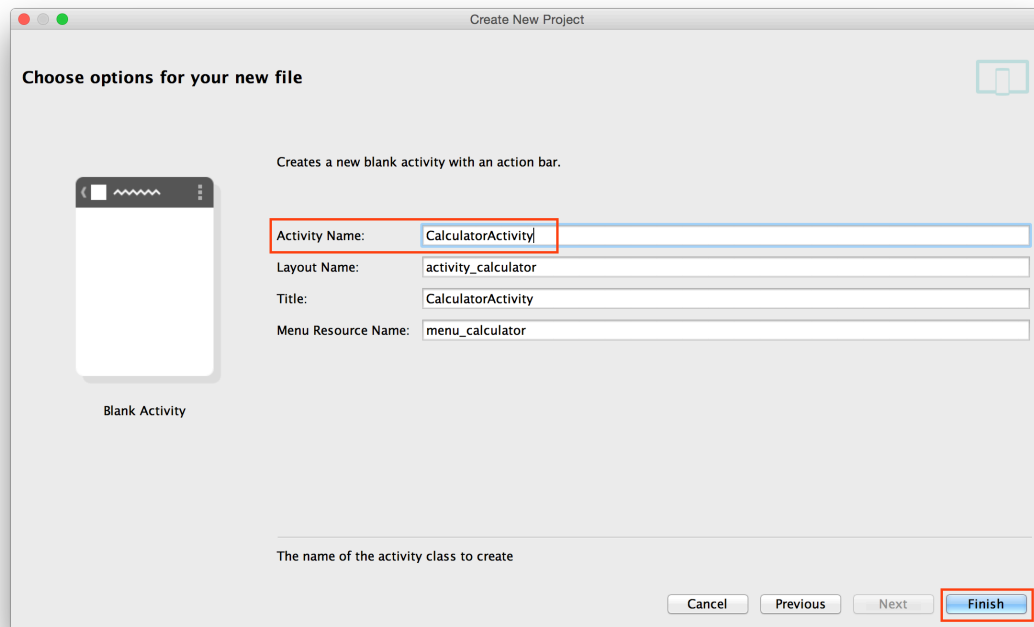
Note: This is the simplest version that still creates an activity for us. We will be using fragments, but we'll add them manually.

As you can see, this screen offers many options for app types you can create in the future.

2.4 File Options

In the last screen we determined what type of files were created. In this screen, we get specific about naming.

Change the default `Activity Name:` to **CalculatorActivity**.



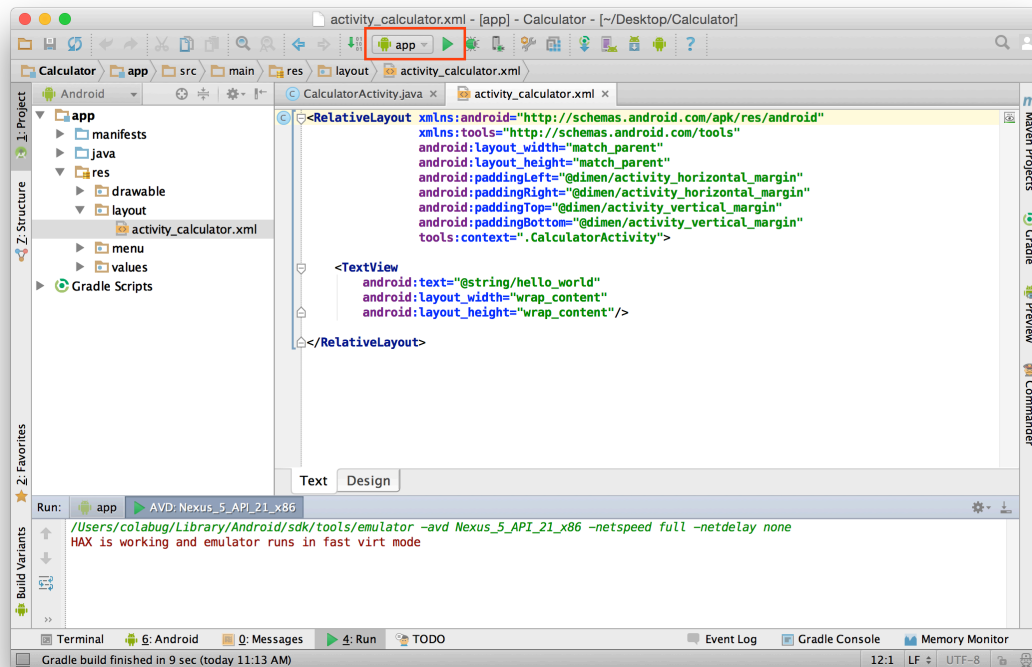
The `Title`, `Layout Name`, and `Menu Resource Name` fields will automatically update based on the name you enter. You can change them, if desired.

This creates an **Activity** called **CalculatorActivity** in the directory:
`app/src/main/java/com/greenlivesoftware/calculator`
Click `Finish`.

2.5 Run

We now have a working application.

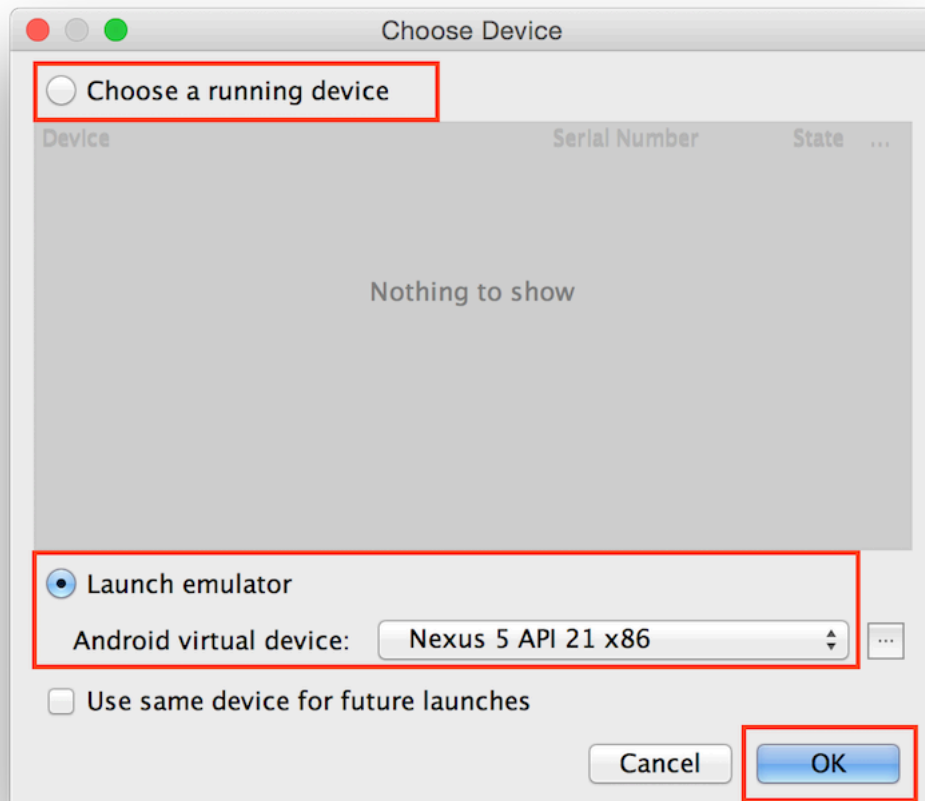
To see it in action, click the play button at the top of the screen (next to the run configuration for `app`).



This opens the “Choose Device” dialog.

Choose Your Device

Here you choose between running the app on an emulator or your own device using the radio buttons.



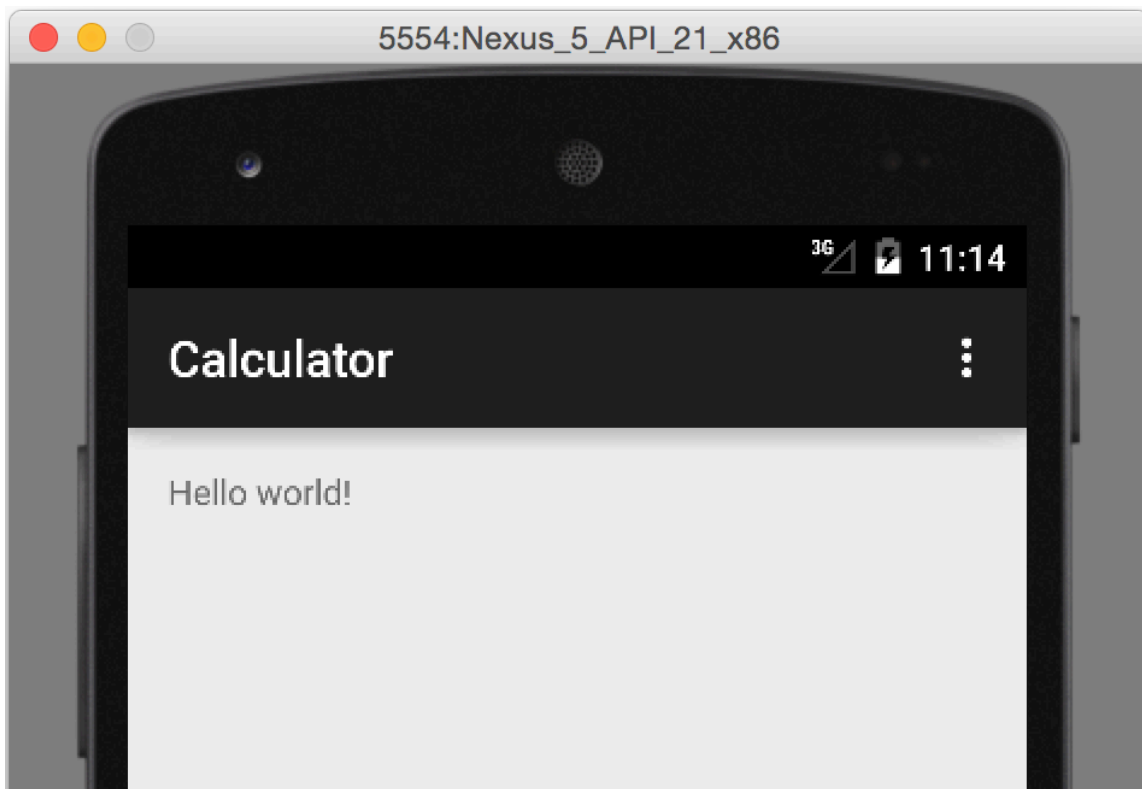
The device list shows emulators you have running or connected devices through USB. You can also launch an emulator that you've previously configured.

Select the checkbox next to `Use same device for future launches` to skip this dialog in the future.

Click `OK`.

Launch Hello World

When the app launches, you see a simple application with welcoming text and an action bar.



2.6 First Commit

Let's clean up a few loose ends and commit our work.

Note: Had we used the `Add No Activity` option in the wizard, we wouldn't have the files we'll remove in the following sections.

Remove the Action Bar

The `ActionBar` and menu were generated by the wizard, but doesn't make sense in a calculator app. We don't have any options or view navigation so it's simply taking up valuable screen real estate.

Let's remove it.

The action bar comes from our app's theme, `AppTheme`, which is defined in `app/src/main/res/values/styles.xml`.

To remove the action bar, replace the current `parent` with `Theme.AppCompat.Light.NoActionBar`.

```
<style name="AppTheme"
    parent="Theme.AppCompat.Light.NoActionBar">
</style>
```

Note: `AppTheme` is specified as the `theme` attribute in the `<application>` tag in `app/src/main/AndroidManifest.xml`.

Remove Options

It's bad practice to leave around code that you aren't using. This is called **dead code** or **unreachable code**. **Technical debt** adds up!

Since we're not using the menu and we've removed the **ActionBar**, there's no way to reach the options.

To remove them, start in the activity.

```
app ▸ src ▸ main ▸ java ▸ com ▸ greenlivesoftware ▸ calculator  
  ▸ CalculatorActivity.java
```

In this file, the menu is inflated in **onCreateOptionsMenu()**.

```
getMenuInflater().inflate( R.menu.menu_calculator, menu );
```

The **inflate()** call references the menu file.

```
app ▸ src ▸ main ▸ res ▸ menu ▸ menu_calculator.xml.
```

Delete the file and its enclosing directory.

When we removed the menu directory, we created two build errors in

```
CalculatorActivity.java. We don't need either of these functions,  
so just delete onCreateOptionsMenu() and onOptionsItemSelected().
```

📁 .gitignore

Finally, there's a few generated files we won't need in the repository.

If we did add them to the repository, each time that we built the application or changed something in the IDE, our changeset would be affected.

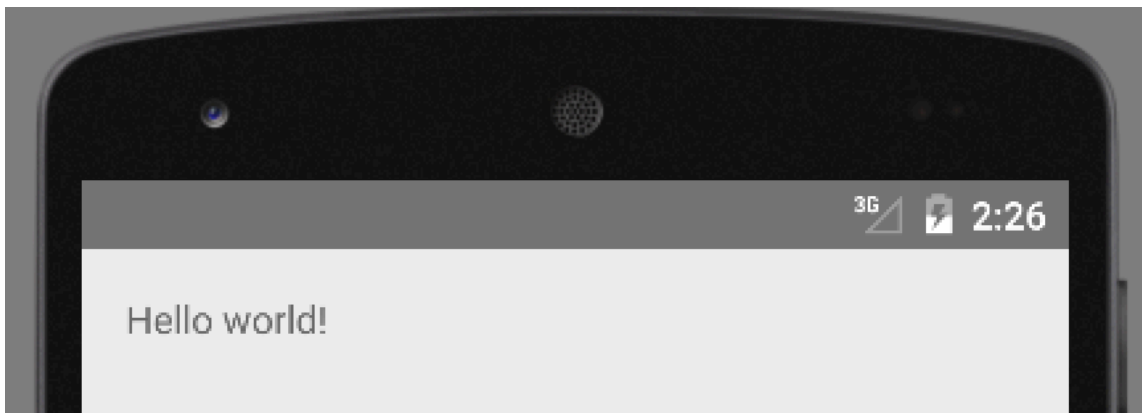
Note: This book assumes git usage, but you are welcome to use your favorite version control system (VCS).

Add these items to your 📁 .gitignore.

```
.gradle
/local.properties
/.idea/workspace.xml
/.idea/libraries
.DS_Store
/build
*.diff
```

Commit

Run the app. The action bar should be gone along with the options button.



Use the IDE integration or command line to prepare your commit.

Chapter 3

Hello, Robolectric

As I mentioned in the [Introduction](#), unit testing and test-driven development (TDD) is an important topic in Android development that is often left to the reader. Instead, we will use it from the beginning so you'll become familiar with it as you build out the application.

We will not always follow a strict TDD flow through out this book. Tasks like integrating libraries and creating helpers are sometimes shown before a test necessitates them. This is a deliberate choice to improve the flow for the reader.

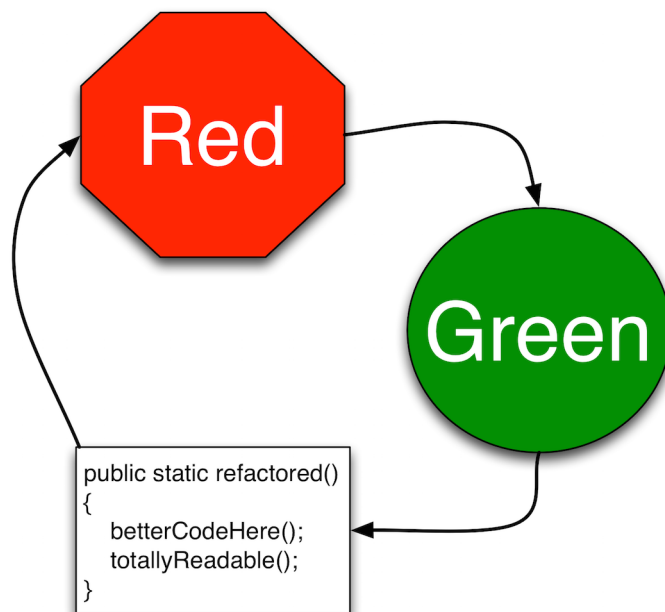
In this chapter, we focus on configuration and tooling. In future chapters, we write a test to verify that a UI element exists (before it does) and then add just enough to make our tests pass.

Note: There's a decent bit of configuration before we can start testing. It may feel like overkill when you're first starting out, but I encourage you to bear with me through the exercises. The first time will feel difficult, but it gets easier and it's much simpler to start a project with Robolectric from the start than to retrofit it later.

TDD

TDD helps you to closely focus on the application you are building piece by piece. You write just enough code to pass a failing test. You can do broad refactoring without worrying about breaking the rest of the system.

TDD is a cycle. You start with a failing test. You make the test pass. Then you refactor, if desired.



At the end of many cycles required to create a feature, you will have a fully tested and implemented feature. When your changes are significant enough and the tests are passing, commit your work.

Robolectric

Unit testing has been particularly difficult in Android. One library that makes this process much simpler is [Robolectric](#).

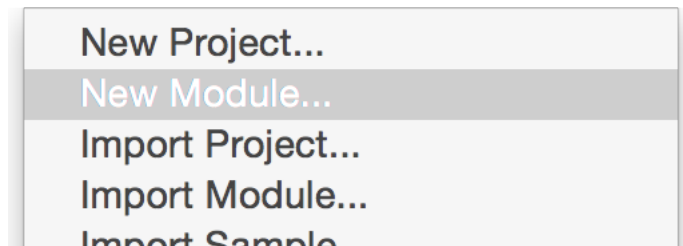
Let's get started.

3.1 Create Test Module

There are plugins out there we could use to configure Robolectric with Android Studio, but we'll take the manual path. This tutorial expands the work of [Paul Blundell](#).¹

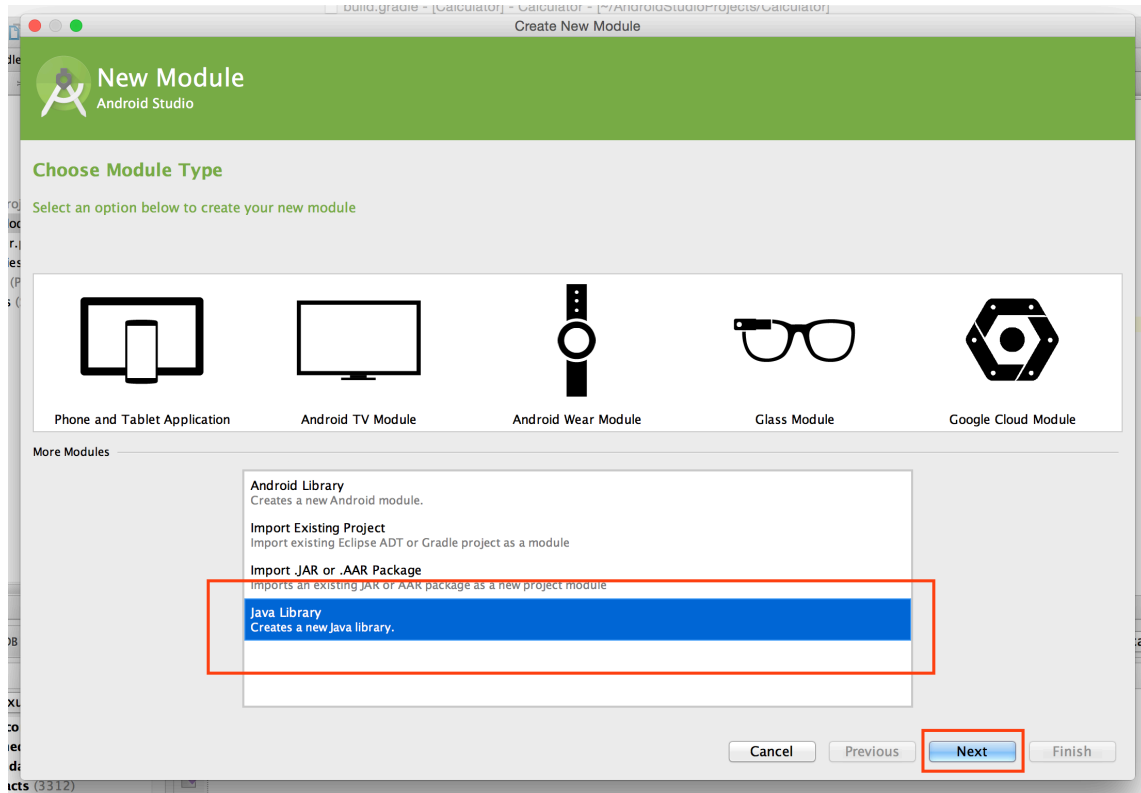
Note: If you want to see a project in action, check out this [repository](#). I created two branches. The **blundell** branch walks through his blog recommendations and implements them with notes along the way. The other branch uses a plugin to configure Android Studio and Robolectric.

First, we'll create a module using Android Studio's tools. Select **File** > **New Module** to start the module wizard.

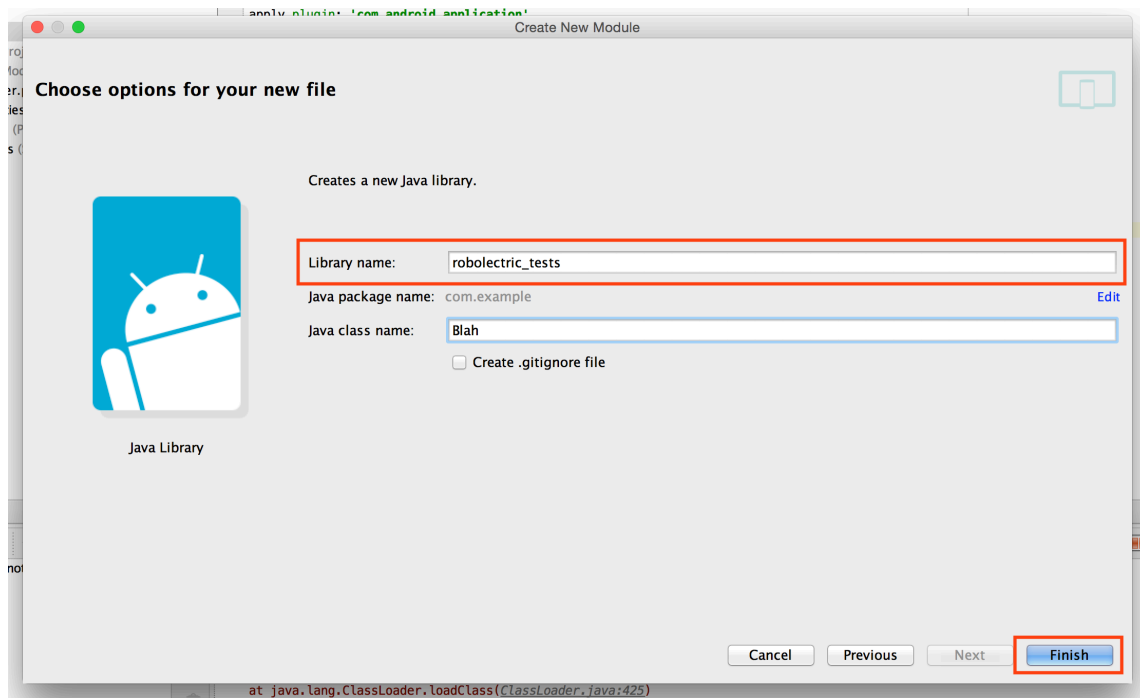


¹Blundell, P. "Android Gradle App with Robolectric JUnit tests." July 12, 2014. Retrieved from [web](#).

From this screen, select **Java Library** which is at the bottom (and possibly off the screen and requires scrolling). Click **Next**.



Now we'll configure the module properties.



Enter “robolectric_tests” as the `Library Name`.

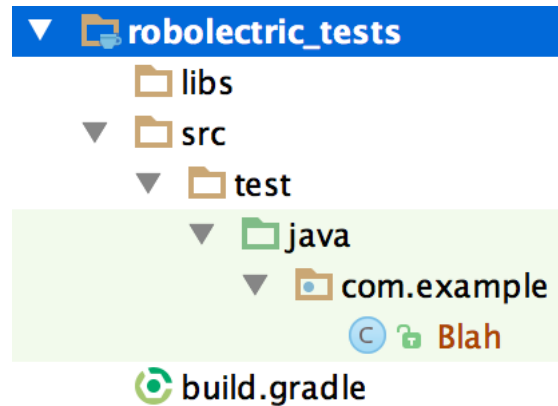
Enter “Blah” in the `Java class name` section to create a placeholder test file (because you can’t skip this step). We’ll delete this file later.

I unchecked `Create .gitignore file` here, but this step is optional. You can maintain a separate ignore file here if that works for you.

Click `Finish`.

Once you are done with the wizard, it will show the main project view.

Expand your new module `roboelectric_tests` (it's a top-level module like `app`). In the module, you'll see the `Blah.java` file that we created with the wizard.



The wizard also created a file called `build.gradle`. In the project view, you'll see it under the `roboelectric_tests` directory.²

If you looked at the file listings from the command line, you'd see this layout:

```
# ls roboelectric_tests
build.gradle
libs
src/test/java/com/example/Blah.java
roboelectric_tests.iml
```

²If you are looking at the Android view, you'll see it in the `Gradle Scripts` section.

Clean Up

We need to change a few things to make the module work seamlessly with the Robolectric testing framework.

📁 test Directory

The wizard put the 📁 Blah.java file in into the 📁 robolectric_tests ▶ src ▶ main ▶ com ▶ example directory.

For testing purposes, we want test classes to live in the 📁 test directory under 📁 src, so we'll rename the 📁 main folder to 📁 test. The final path should be: 📁 robolectric_tests ▶ src ▶ test ▶ com ▶ example.

Unfortunately you can't easily see the full path in Android Studio, so you will likely need to do this step in a file explorer or on the command line.

Delete 📁 Blah.java

Now we'll delete both the file 📁 Blah.java and the directory 📁 example that contains it.

However, we want to keep the directory structure below it intact. This will allow us to easily create test files in the right places by using IDE shortcuts.

📁 androidTest Directory

When we used the wizard to create this project, it created a test file: 📁 app ▸ src ▸ androidTest ▸ java ▸ com ▸ greenlivesoftware ▸ calculator ▸ ApplicationTest.java.

This file isn't needed for this project and causes issues with the tests. When we use shortcuts to create the tests, it will put it in the wrong folder while this one exists.

Delete the directory 📁 androidTest and all files underneath it.

Commit

Now that we have the module configured, it's time to save our work.

Let's add a file called 📁 .gitkeep to the directory 📁 robolectric_tests ▸ src ▸ test ▸ java ▸ com ▸ greenlivesoftware so that git will add this path to the working tree even though the directory is empty.

To do this right click on `com.greenlivesoftware`, select **New** > **File** and add a text file named 📁 .gitkeep.

The 📁 build directory is generated in each module that uses Gradle during the build process. To exclude this from your repository (because it's noisy and generated), add the line `robolectric_tests/build/*` to the top-level 📁 .gitignore file.

Now we're ready to commit.

3.2 Integrate Robolectric

The next step is to integrate the Robolectric toolchain into our project.

Add Dependencies

To add the dependencies, open the `build.gradle` file in the `roboelectric_tests` module.

Leave the top line alone and replace the dependencies with the latest version of Robolectric (2.4 at the time of writing) and junit. We exclude the support library from Robolectric because it causes build errors and is not needed for testing.

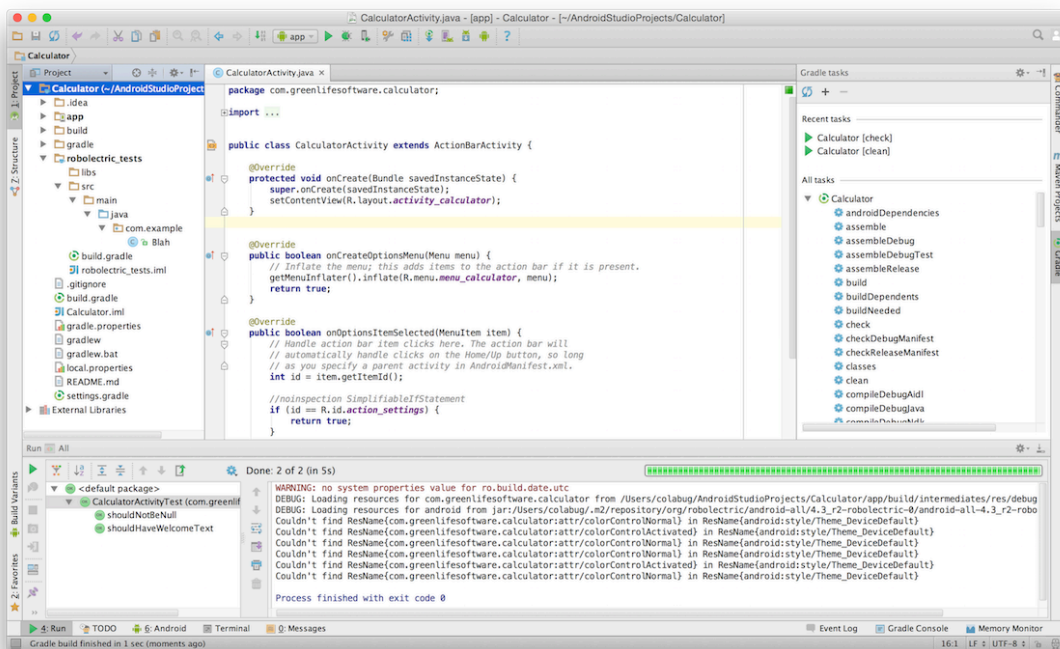
```
apply plugin: 'java'

dependencies
{
    testCompile 'junit:junit:4.+'
    testCompile('org.robolectric:robolectric:2.4')
    {
        exclude module: 'support-v4'
    }
}
```

Summary

This is a very important chapter because we will use unit testing throughout this book to help us build out the UI. We:

- Integrated the industry-leading unit testing tool, Robolectric¹⁷, and wrote two passing tests.
- Ensured the integration worked by running our sanity check on the activity.
- Created run configurations and learned a few ways to build the project.
- Ensured resources were configured for our welcome view.
- Created utilities for testing resources used in our UI.



¹⁷Robolectric is used in many Android coding teams. If they don't use it now, you can become a leader and advocate for higher quality code.

Chapter 10

Constructing Numbers

In this chapter, we add the logic for consecutive numbers appending to the currently displayed number – up to the maximum digit length.

10.1 Handling Append Events

We'll start by ensuring that a number event triggers an append event.

The current logic¹ in `CalculatorStateFragmentTest.java` is to catch a `NumberEvent` and reroute as a `DisplayEvent`. Instead we'd like to send a more specific `AppendEvent` in its place.




First, rename the test to `numberEventShouldFireAppendEvent()`.

```
@Test
public void numberEventShouldFireAppendEvent() throws Exception
{
    String NUMBER_VALUE = "1";
    bus.post( new NumberEvent( NUMBER_VALUE ) );

    BaseEvent lastEvent = busHelper.getLastEvent();
    assertTrue( lastEvent instanceof AppendEvent );
    assertEquals( ((DisplayEvent) lastEvent).getValue(),
                  equalTo( NUMBER_VALUE ) );
}
```

¹We migrated a test and logic for rerouting events from the activity to the fragment in the [Migrate Activity State](#) section.

Promote **NUMBER_VALUE** to a constant field using  +  + .


Next, create a helper function named **postNumberEvent()** to make the intention of our code clear. To do this, highlight the bus posting line and use  +  +  to create a method.

```
private void postNumberEvent()  
{  
    bus.post( new NumberEvent( NUMBER_VALUE ) );  
}
```

After these refactors, the tests should still pass.

Finally, change **DisplayEvents** to **AppendEvents** in the test. s

```
@Test  
public void numberEventShouldFireDisplayEvent() throws Exception  
{  
    postNumberEvent();  
  
    BaseEvent event = busHelper.getLastEvent();  
    assertTrue( event instanceof AppendEvent );  
    assertEquals( ((AppendEvent) event).getValue(),  
        NUMBER_VALUE );  
}  
  
private void postNumberEvent()  
{  
    bus.post( new NumberEvent( NUMBER_VALUE ) );  
}
```

This creates a build error, so use the shortcut  +  from the test to create `AppendEvent.java` in the `events` package.

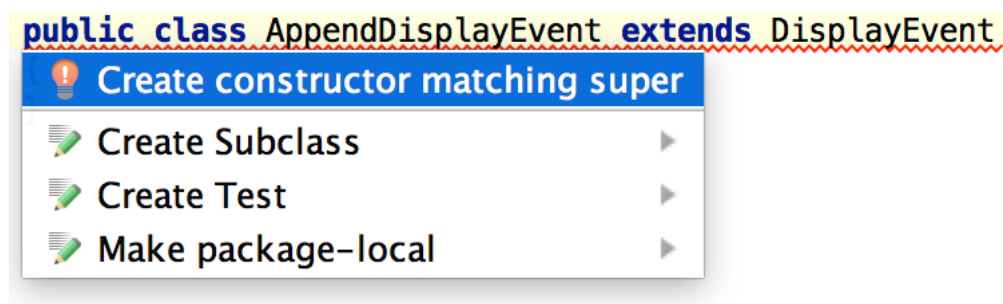
Append Display Event

This event will tell the display to append to the currently shown value.

First, extend **DisplayEvent**.

Once you've done that, the class complains that there is no constructor matching super. This is because when we created `DisplayEvent.java`, we only created one constructor that takes a value.

Use the key combination `⌘`+`⌘` to fix it.



Your class should look like this:

```
public class AppendEvent extends DisplayEvent
{
    public AppendEvent( String value )
    {
        super( value );
    }
}
```

Now run all tests in `CalculatorStateFragmentTest.java`. The first assertion fails (that it's an instance of **AppendEvent**) because we need to update the rerouting logic.

Handle Append Events

In `CalculatorStateFragment.java`, our `onNumberSelected()` subscription reroutes `NumberEvents` to `DisplayEvents`. Simply update the subscription to post an `AppendEvent` instead.

```
@Subscribe
public void onNumberSelected( NumberEvent event )
{
    CalculatorApplication.postToBus( new AppendEvent( event.getNumber() ) );
}
```

We will add logic starting in the [Set Display After Operator](#) section to determine when to append and when to replace the display text.

Rerun the tests. They should pass.

10.2 Append the Display

Add Test

In `DisplayFragmentTest.java`, add a test for updating the displayed value named `appendEventShouldAppendDisplay()`.

In the test, set the starting value for the display and post an `AppendEvent` to the bus. Finally, assert that the text has been appended to the end.

```
@Test
public void appendEventShouldAppendDisplay() throws Exception
{
    display.setText( TEST_VALUE );
    bus.post( new AppendEvent( TEST_VALUE ) );
    assertThat( display.getText().toString(),
                equalTo( TEST_VALUE + TEST_VALUE ) );
}
```

Run the tests. They fail with this error:

```
java.lang.AssertionError:
Expected: "TestTest"
but: was "Test"
```

At this point, we're not appending text, simply replacing. Let's change that.

Append the Display

In `DisplayFragment.java`, add a subscription for **AppendEvents**. In the function, get the currently displayed value, append the string from the **event** and set the display.

```
@Subscribe
public void onAppendDisplay( AppendEvent event )
{
    setDisplay( getDisplayString() + event.getValue() );
}
```

Use the shortcut `⌘ + ↵` to generate the **getDisplayString()** helper function. This returns the string that's currently configured in the view.

```
public String getDisplayString()
{
    return getDisplayView().getText().toString();
}
```

Use the shortcut `⌘ + ↵` again to generate **getDisplayView()** that fetches a reference to the **EditText**.

```
private EditText getDisplayView()
{
    return (EditText) layout.findViewById( R.id.calculator_display );
}
```

Finally, use the **getDisplayView()** helper to simplify **setDisplay()**.²

```
private void setDisplay( String displayString )
{
    getDisplayView().setText( displayString );
}
```

²Typically this would be done after the tests are in a passing state. It's here for narrative flow.

17.1 About the Author

[Corey Leigh Latislaw](#) began her programming career with a different focus than most. She is guided by a passion for traveling the world, bridging the digital divide, advocating for environmental causes, and eating the best food the world has to offer. She uses her skills for the greater good, to improve the lives of others.

She began encouraging broader participation of women and minorities in STEM careers while studying computer science at Florida State University. She was a leader in both the [ACM-W](#) and [STARS Alliance](#) organizations. She continued this work with Women in Cable Telecommunications of Philadelphia where she served on the board and created events such as Tech Camp 2.0, a conference that explored the future of cable technology.

She continues to pursue this passion and serves as VP of Operations on the board of [Kids on Computers](#), which opens computer labs in developing nations. This June she worked in [Huajuapán de León](#) and plans to attend future trips with the organization. She also leads local workshops through the [Android Alliance](#), GirlDevelopIt, and TechGirlz organizations.

She began her mobile career at Comcast Interactive Media. She helped build the XfinityTV Android & iOS applications from the ground up. While there, she founded the [Philadelphia Android Alliance](#). This organization connected Android developers and created many strong speakers from the region. It has evolved into the Google Developer Group of Philadelphia and organizes many meetups on various Google technologies.

She continued writing Android applications with local firms for a few years and then set out on her own to run a [training and consulting company](#). She's passionate about the Android ecosystem and can't wait to see where the platform takes us in the coming years.

She has become a sought after [international speaker](#) that has traveled near and far to teach Android as well as discuss programming, technology trends, security, open source, and feminism.