

Android Activity Book, Beta 1.2

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.” It 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. New chapters reinforce material learned from previous ones.

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 in which we write applications.

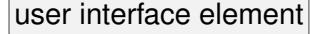
Only through practice will you become comfortable. When you come back to this book the second, third, or fourth time your muscle memory will take over. You will achieve programming flow while creating apps quickly and efficiently with a high level of confidence in their correctness.

Using the book this way will help you become comfortable with tooling integration, keyboard shortcuts, generating code with your IDE, 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: 

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

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

Shortcuts

Keyboard shortcuts are the lifeblood of the developer! Let the computer do some work for you.

Shortcuts reduce boilerplate code and make it much more enjoyable to write software. You will move faster the more you get away from your mouse as the major way to interact.

The shortcut you will use the most is `⌘+⌫`. In many places throughout the book, we create a build error on purpose and solve it with this key combination. You can use it in XML and Java alike.³ It can be used to solve so many ailments that it deserves a category of its own. If you take away nothing else, remember this!

³It doesn't always do everything that you expect, but something weird happens, use `⌘+Z`.

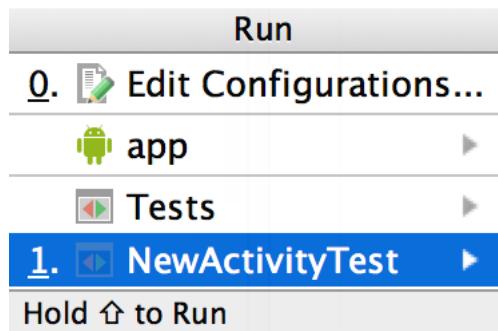
Testing

Code Generation

- `⌘ + ⌘ + T` - Create a test file or open existing
- `⌘ + N` - Generate test or setup

Run Configurations

- `ctrl + R` - Run the last run configuration
- `ctrl + D` - Debug the last run configuration
- `⌘ + ⌘ + R` - Open run configuration window



Coding

Copying

- + - copy entire line
- + - duplicate entire line
- + - cut entire line

Generation

- + Override class methods
- + Generate code (context aware)

Refactor

- + - Change signature
- + - Change name
- + - Move statement or method up (reorder)
- + - Move statement or method down (reorder)
- + Comment/Uncomment selection

Abstract

Select the code that you'd like to refactor by using the mouse or by selecting scope with + .

- + + Create method

-  +  +  Create variable
-  +  +  Create field
-  +  +  Create constant field

Formatting

-  +  +  - Reformat code

Navigation

Navigation

There are 3 ways to open a file:

- Use + + and type the name,
- Use + while the cursor is on the file name, or
- Hold and click on the class name.

Search

- + Find in file
- + Replace in file
- + + Find in project

Check out the improving productivity guide at the end of this book for more ideas.

Introduction

We will build a calculator through a series of exercises.⁴ 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](#).⁵

⁴Code samples for the book will be posted at [github](#).

⁵I'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 re-structure 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
Shortcuts	v
Introduction	x
Contents	xii
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

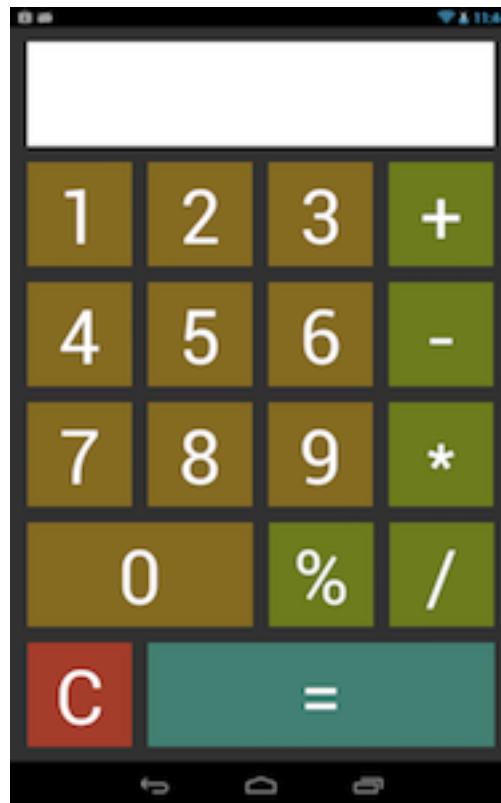
7.7 Equals Button	128
8 Updating the Display	137
8.1 Routing Calculator Events	138
8.2 Add Application & Test	140
8.3 Setup Event Bus	142
8.4 Number Events	144
8.5 Post Number Event	149
8.6 Update Number Display	153
8.7 Update Operator Display	162
9 Calculator State	192
9.1 Adding State	193
9.2 Calculator 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	292
13 Computing Results	318
14 Error States	319
15 Scaling the View	320
15.1 Flexible Dimensions	320
15.2 Device Rotation	322
15.3 Landscape	322
16 Appendices	323
16.1 Glossary	323
16.2 Resources	324
16.3 Improving Productivity	325
17 Epilogue	327
17.1 About the Author	327

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

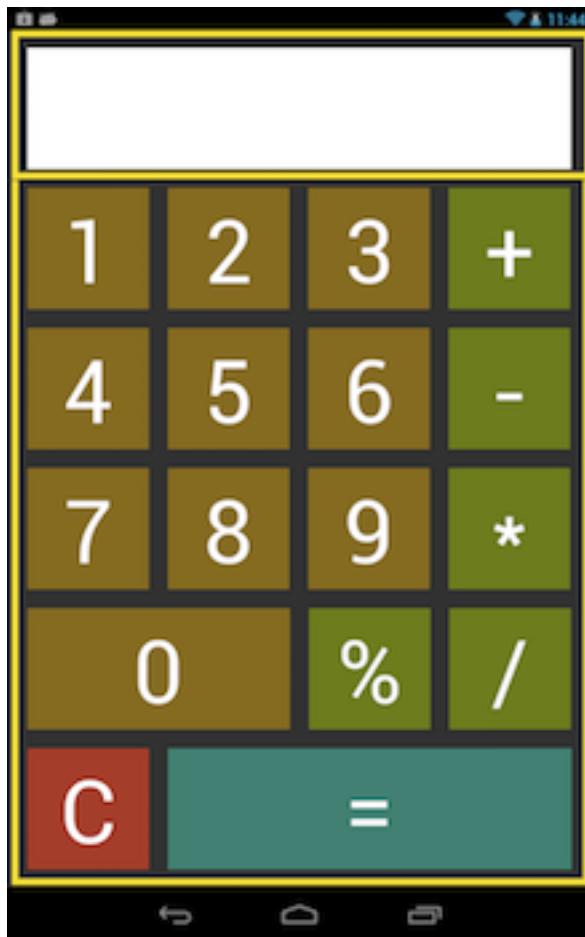
How would you break down the view? What [view elements](#) would you use?

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 **FragmentManager**s laid out in a **LinearLayout** with vertical orientation. We'll cover this more in depth in [Chapter 4](#) and [Chapter 5](#).



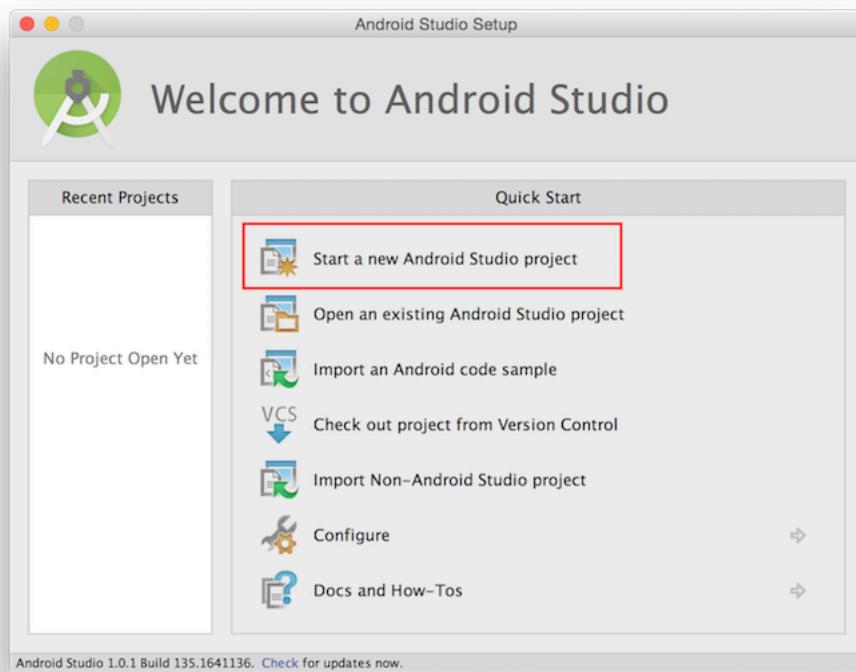
The **DisplayFragment** will handle the display. We'll use an **EditText** to display the numbers entered into the calculator as well as calculations.

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

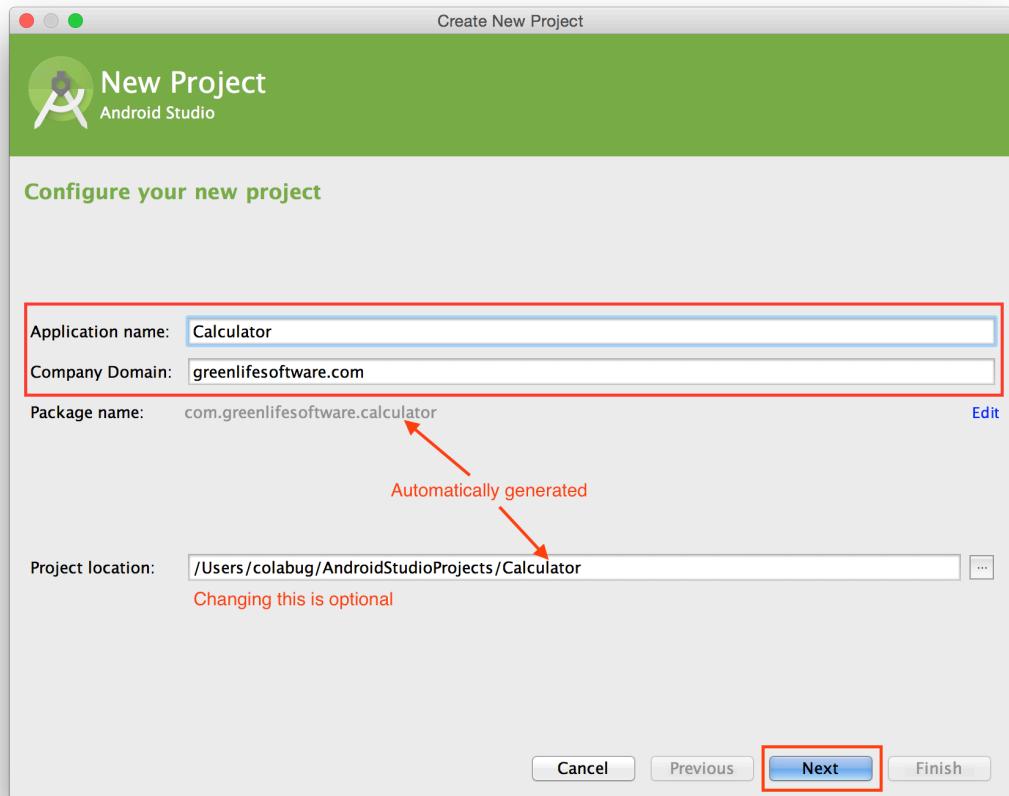
We'll use Android's project templates to create our application. From the welcome screen, select **Quick Start > Start a new Android Studio project.**. From inside the application, use **File > New Project....**.



2.1 Project Wizard

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

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



The combination of the application name and the company domain generates the Package name: **com.greenlifesoftware.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 greenlifesoftware.com.

For future projects you plan to release, if you don't have a domain, you can buy one or create a [github](#) 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.

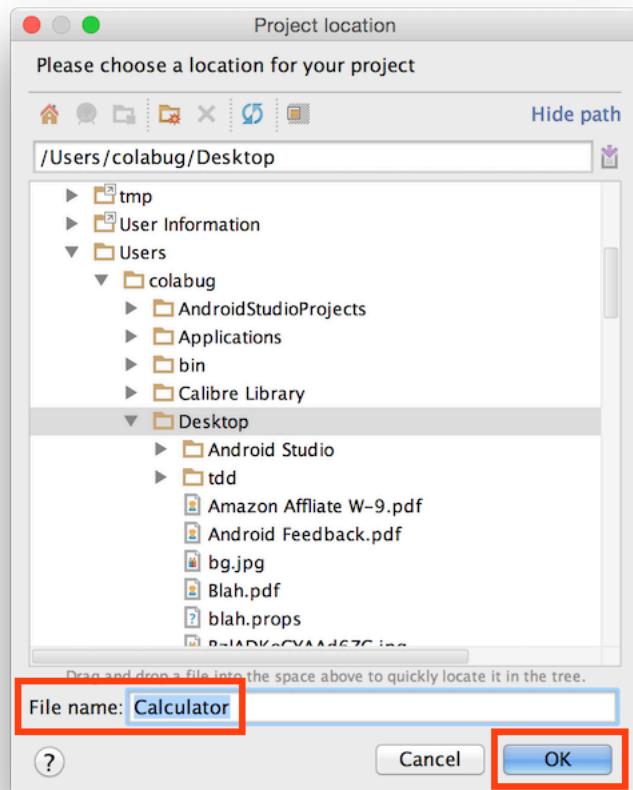
This means your current users would need to install a different application from the market instead of getting a notification that an update is available.¹ This is not only a pain for your users, but could damage your brand and lower your discoverability.

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.



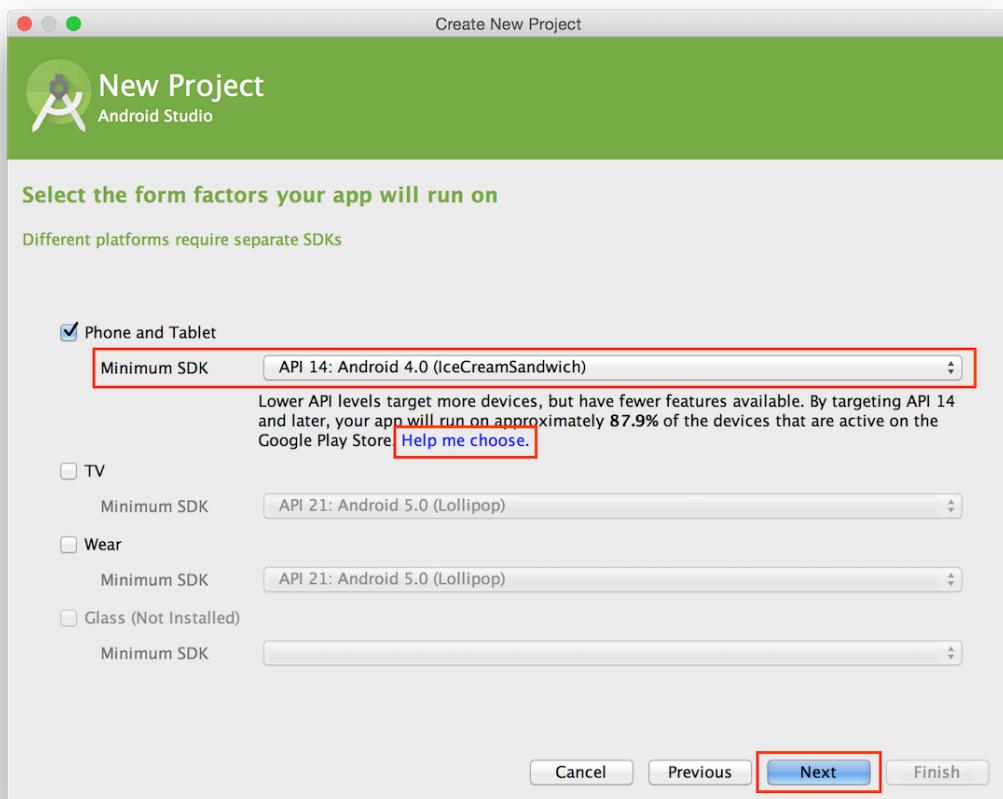
Default Mac location: **Macintosh HD** ▶ **Users** ▶ [user] ▶ **AndroidStudioProjects**

Click **OK** when done.

2.2 Choose Form Factors

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

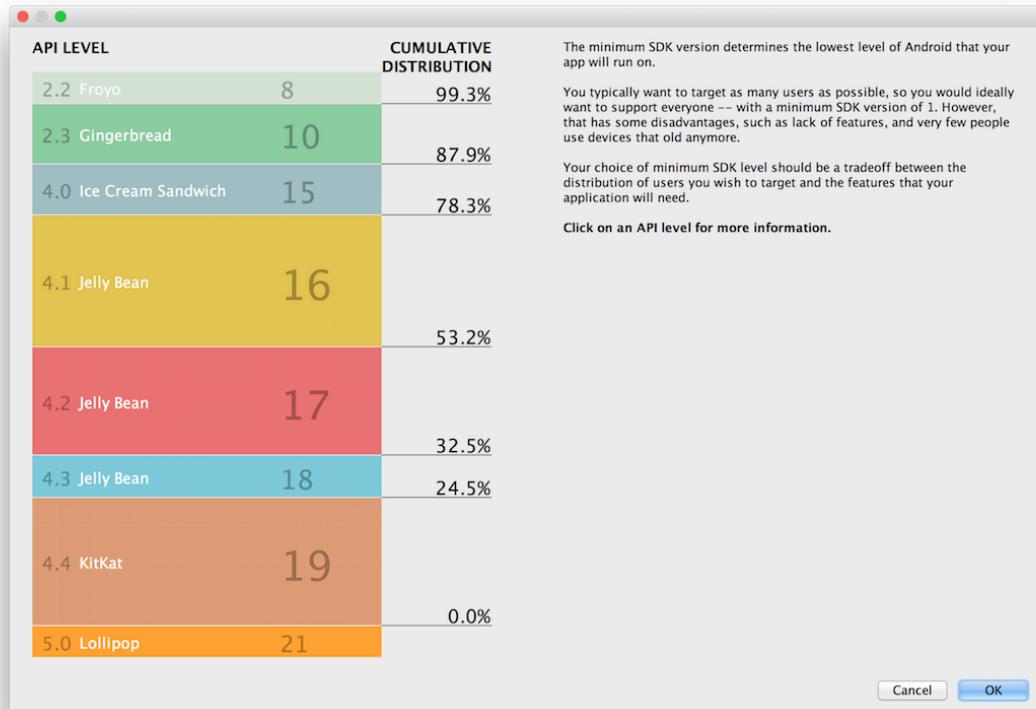
Check the **Phone and Tablet** box and select **API 14: Android 4.0 (IceCreamSandwich)**² from the **Minimum SDK** drop-down. Click **Next**.



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

²This 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](#).

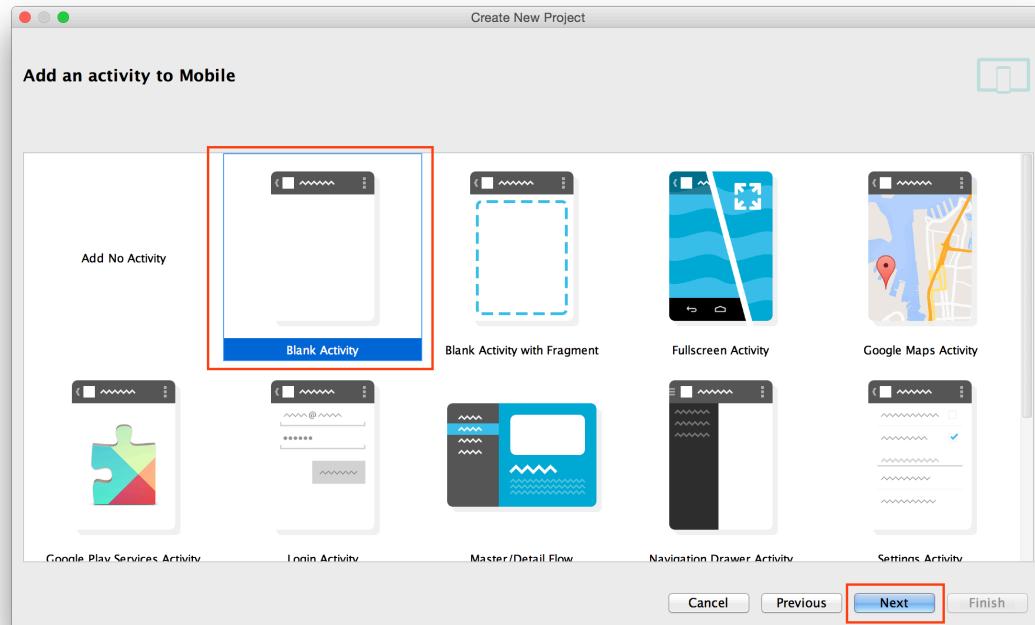


Choosing an appropriate API for your project is a subject that requires several inputs to determine the best one to suit your needs. The choice is influenced by your target demographic, API-specific features, devices you plan to support, current trends, and market share among other considerations.

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**.

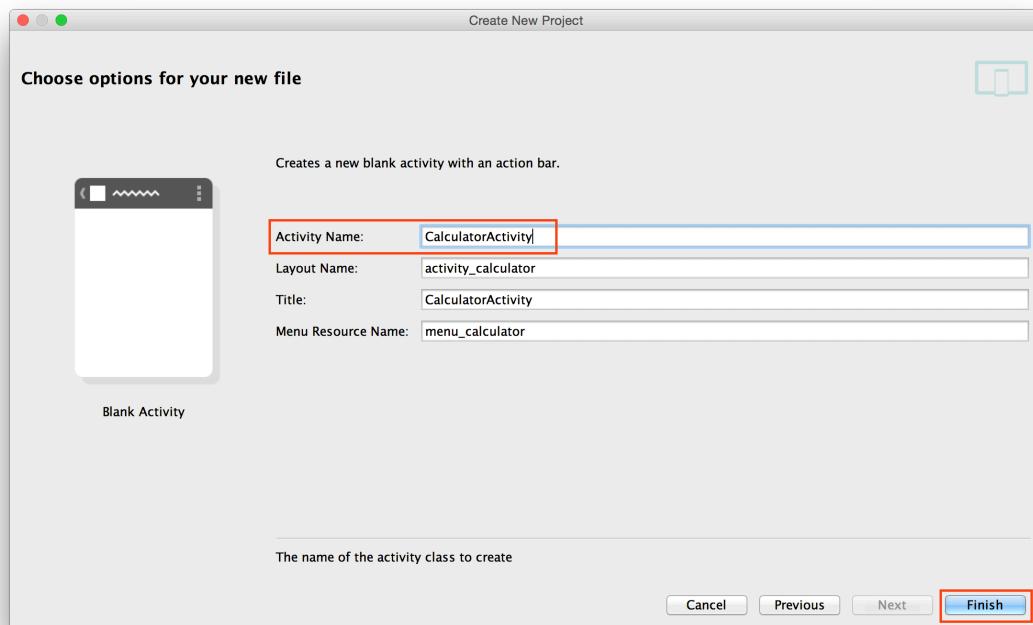
Note: This is the simplest version that still creates an activity for us. We will be using fragments, but we'll add them manually.

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.

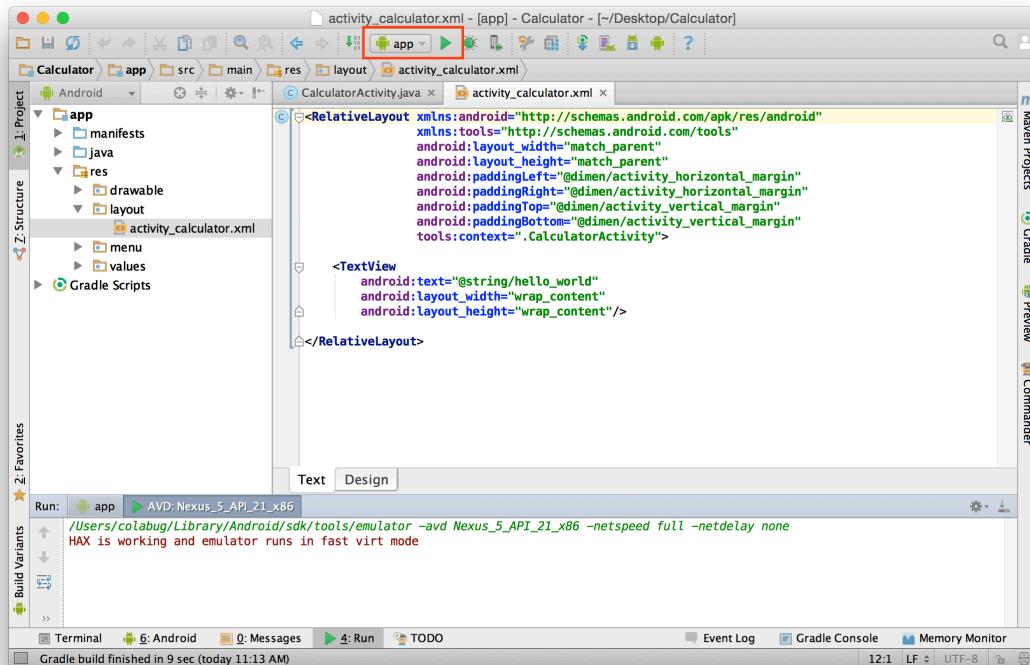
This creates an **Activity** called **CalculatorActivity** in the directory:
app>src>main>java>com>greenlifesoftware>calculator

Click **Finish**.

2.5 Run

We now have a working application.

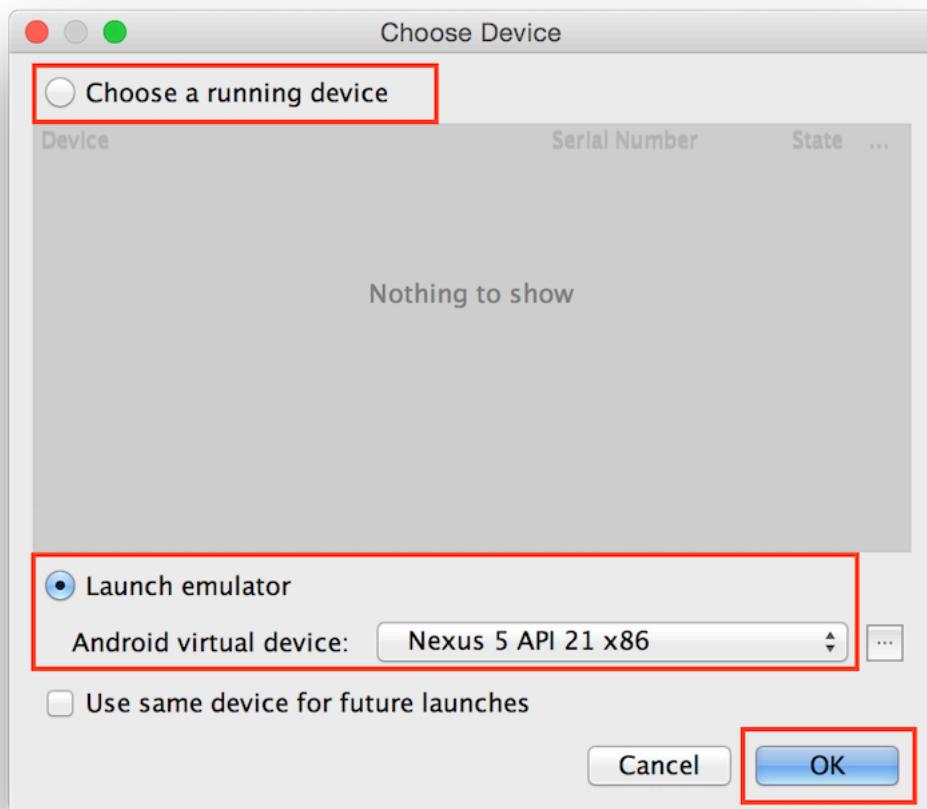
To see it in action, click the play button at the top of the screen (next to the run configuration for `app`).



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.



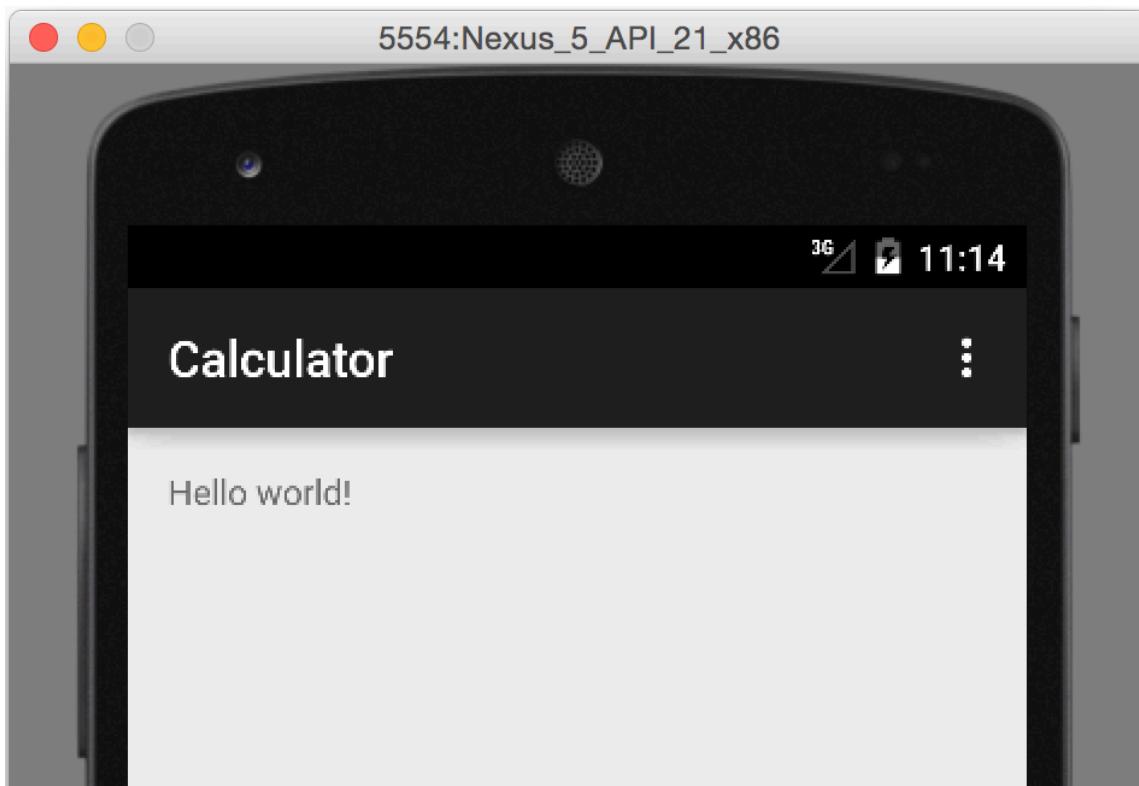
The device list shows emulators you have running or connected devices through USB. You can also launch an emulator that you've previously configured.

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 that 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>greenlifesoftware>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.

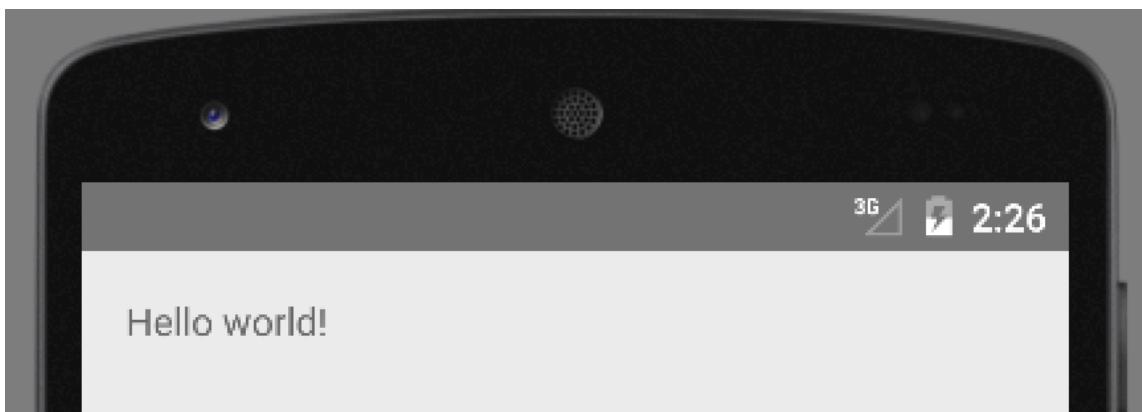
Note: This book assumes git usage, but you are welcome to use your favorite version control system (VCS).

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.

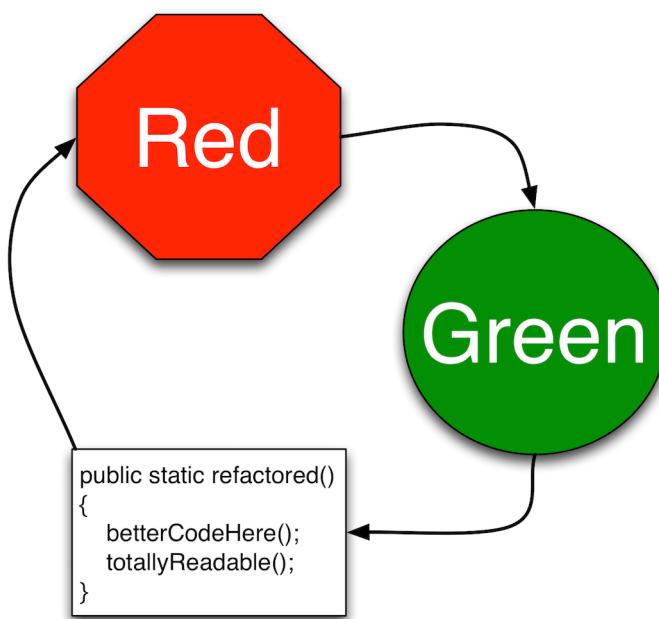
In this chapter, we focus on configuration and tooling. In future chapters, we write a test to verify that a UI element exists (before it does) and then add just enough to make our tests pass.

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](#).

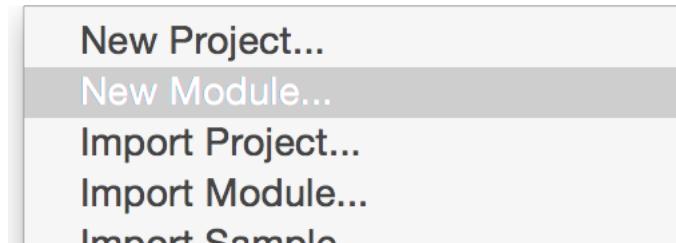
Let's get started.

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

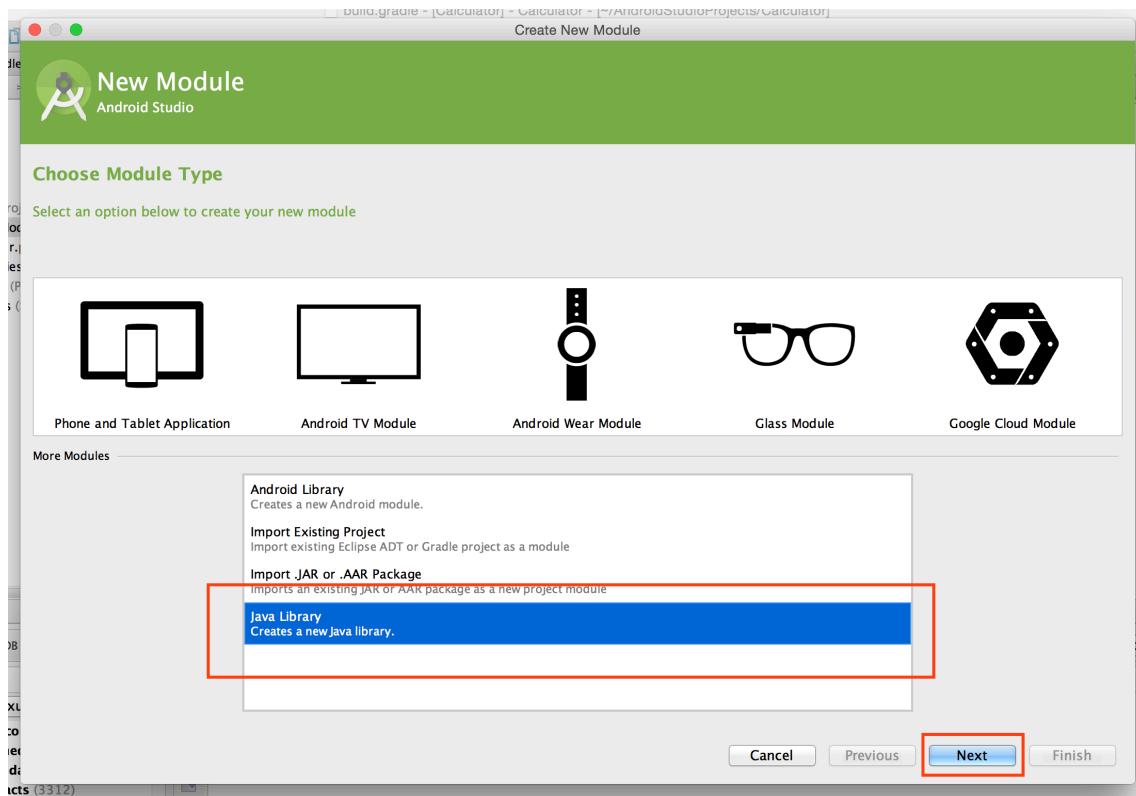
Note: If you want to see a project in action, check out this [repository](#). I created two branches. The **blundell** branch walks through his blog recommendations and implements them with notes along the way. The other branch uses a plugin to configure Android Studio and Robolectric.

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

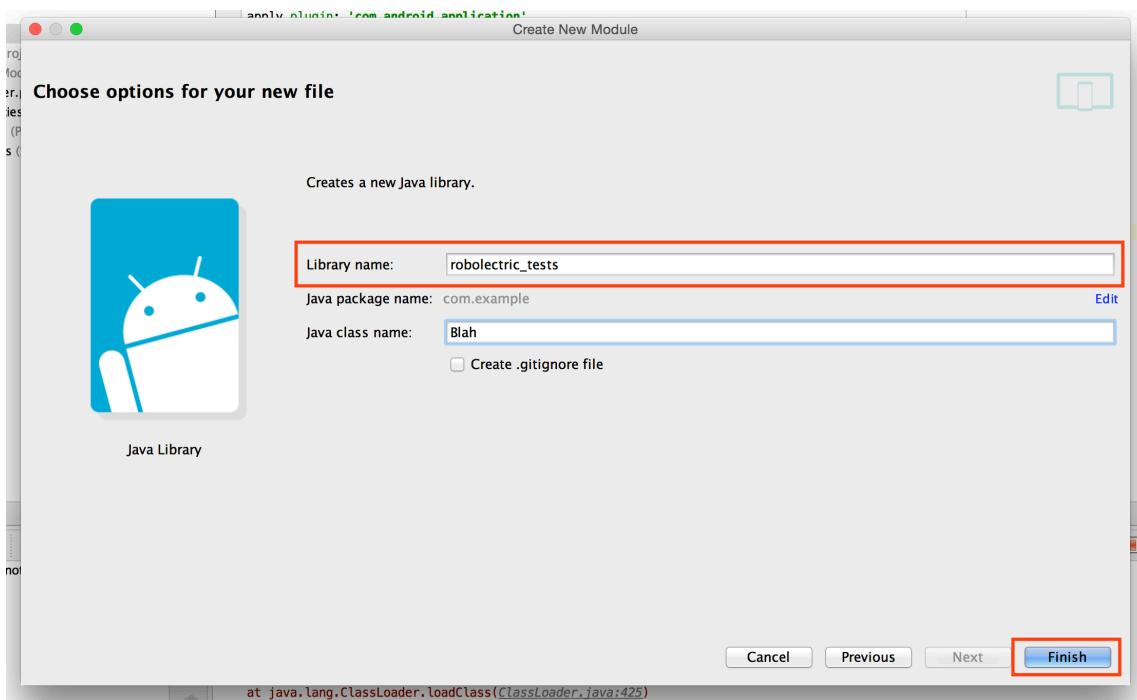


¹Blundell, P. “Android Gradle App with Robolectric JUnit tests.” July 12, 2014. Retrieved from [web](#).

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**.

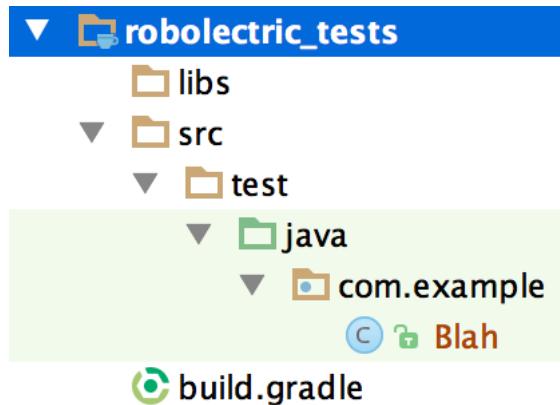
Enter “Blah” in the **Java class name** section to create a placeholder test file (because you can't skip this step). We'll delete this file later.

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 `robolectric_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 `robolectric_tests` directory.²

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

```
# ls robolectric_tests
build.gradle
libs
src/test/java/com/example/Blah.java
robolectric_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 ➤ greenlifesoftware ➤ 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 ➤ greenlifesoftware so that git will add this path to the working tree even though the directory is empty.

To do this right click on `com.greenlifesoftware`, 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 `robolectric_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'
    }
}
```

Associate with app

Our new module doesn't know where to find our application's code. To do that, we edit the `build.gradle` file in the `robolectric_tests` module (same file we edited in the last step).

Add a variable `androidModule` to fetch the project information from the `app` module. After that, we add a step to compile the project to make. This ensures that we are operating on the latest version of our code base when running the tests.

```
dependencies
{
    def androidModule = project(':app')
    compile androidModule

    testCompile androidModule.android.applicationVariants
                .toList().first().javaCompile.classpath
    testCompile androidModule.android.applicationVariants
                .toList().first().javaCompile.outputs.files
    testCompile files(androidModule.plugins
                    .findPlugin("com.android.application")
                    .getBootClasspath())
    ...
}
```

The next few lines mean ...

3.3 Custom Test Runner

According to the [docs](#), **RobolectricTestRunner**:

Creates a runner to run testClass. Looks in your working directory for your `AndroidManifest.xml` file and `res` directory by default. Use the **Config** annotation to configure.

Since the working directory³ is not where we should be looking for the project's manifest, we need to create our own test runner and point it to the manifest.

Note: This is not required for library projects because they don't need the manifest or resources.

We need to do this because typically the `app` module and `robolectric_tests` module would be together instead of separate top-level modules.⁴

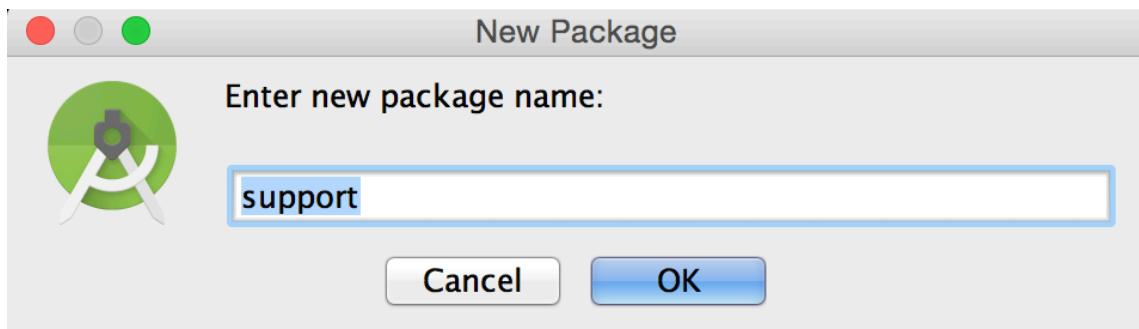
³What *is* the working directory? That's vague.

⁴In theory, we should be able to use a simple configuration file, but I have not been able to get that option to work.

Create support package

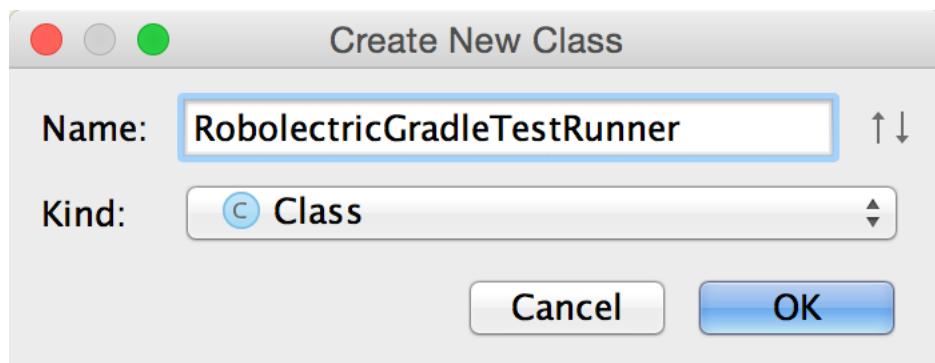
This directory will be used to hold files that support the tests (not the tests themselves).

Right click on the package name `greenlifesoftware.calculator` select to `New > Package` and then enter “support.”



Create RobolectricGradleTestRunner.java

Create a file that extends the builtin test by right clicking on the  support folder and selecting `Java Class`. Enter the class name.



Now extend `RobolectricTestRunner` when declaring the class.

Our class is rather boring, but already has a compilation error. If you hover over the red squiggly lines, it will show you this message:

```
public class RobolectricGradleTestRunner extends RobolectricTestRunner {  
}  
There is no default constructor available in 'org.robolectric.RobolectricTestRunner'
```

Use  +  (best shortcut, memorize this) to create a constructor for you:

```
public class RobolectricGradleTestRunner extends RobolectricTestRunner  
{  
    public RobolectricGradleTestRunner(Class<?> testClass)  
        throws InitializationError  
    {  
        super(testClass);  
    }  
}
```

Now that we have a class that compiles, we'll need to override a few things in order to make it work.

Fetch Manifest

We override the function `getAppManifest()` to specify where to fetch the manifest.

We construct the paths for the manifest and the resources and then construct a new `AndroidManifest` object with these paths.⁵

When we construct the `AndroidManifest` object, we override the `getTargetSdkVersion()` function to limit the SDK version we're using because Robolectric doesn't support higher versions than 18 at the time of writing.⁶

```
// Max SDK version
private static final int MAX_SDK_SUPPORTED_BY_ROBOLECTRIC = 18;

// Project locations
private static final String MANIFEST_PATH = "src/main/AndroidManifest.xml";
private static final String RES_PATH = "build/intermediates/res/debug/";

// Path configuration
private static final String PROJECT_DIR = getProjectDirectory();
private static final String FINAL_MANIFEST_PATH = PROJECT_DIR + MANIFEST_PATH;
private static final String FINAL_RES_PATH = PROJECT_DIR + RES_PATH;

@Override
public AndroidManifest getAppManifest(Config config)
{
    return new AndroidManifest(Fs.fileFromPath(FINAL_MANIFEST_PATH),
                               Fs.fileFromPath(FINAL_RES_PATH))
    {
        @Override
        public int getTargetSdkVersion()
        {
            return MAX_SDK_SUPPORTED_BY_ROBOLECTRIC;
        }
    };
}
```

⁵Custom test runner adapted from the [gradle-robojava-plugin](#) by Gautam Korlam.

⁶TODO: Link to the github issue.

If the editor doesn't do it automatically for you, you can import the classes using the shortcut .



Now all that's left to do is define the function `getProjectDirectory()` which returns the absolute paths needed. First we get the current working directory with `file.getCanonicalPath()` and then manipulate strings with `replace()` to point to the correct directories.

```
// File handling
private static final String EMPTY_STRING = "";
private static final String PATH_SEPARATOR = "/";
private static final String CURRENT_DIRECTORY = ".";

// Project modules
private static final String APP_MODULE = "app";
private static final String ROBOLECTRIC_MODULE = "robolectric_tests";

private static String getProjectDirectory()
{
    String path = EMPTY_STRING;

    try
    {
        File file = new File(CURRENT_DIRECTORY);
        path = file.getCanonicalPath();

        // Test module
        path = path.replace(ROBOLECTRIC_MODULE, EMPTY_STRING);

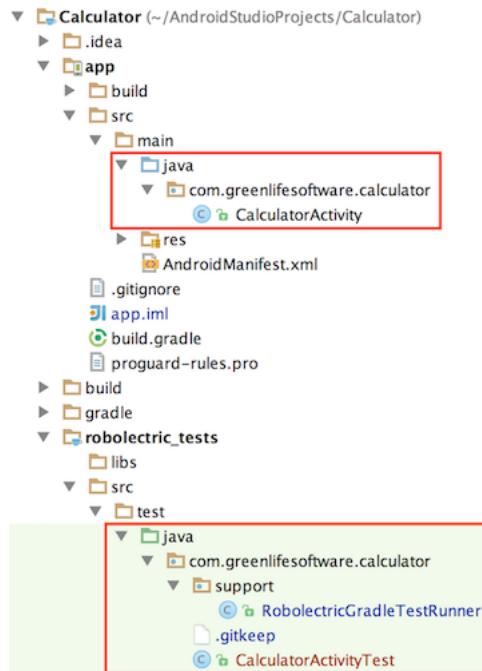
        // Application module
        path = path.replace(APP_MODULE, EMPTY_STRING);
        path = path + PATH_SEPARATOR + APP_MODULE + PATH_SEPARATOR;
    } catch (IOException ignored)
    {
    }

    return path;
}
```

Now we're all set to test.

3.4 First Test

In this lab, we'll create our first test. These tests will live in a parallel folder structure that mimics your source files:



For example, `CalculatorActivity.java` is in the `app` module and `CalculatorActivityTest.java` is in `robolectric_tests`, but both are in the `com.greenlifesoftware.calculator` package.

On disk you can find these files here:

`app>src>main>java>com>greenlifesoftware>calculator>.java`

`robolectric_tests>src>test>java>com>greenlifesoftware>calculator>.java`

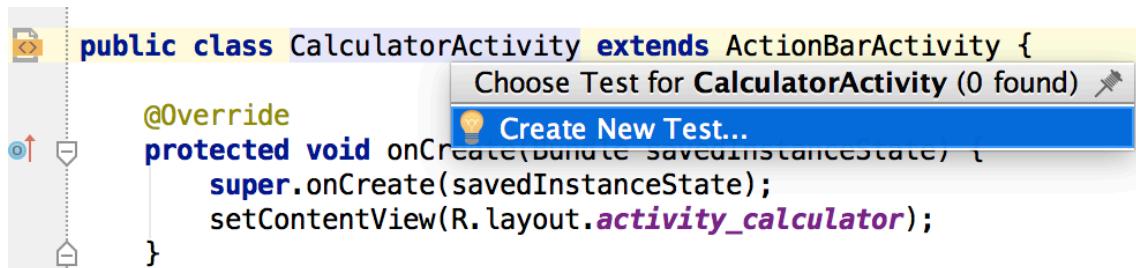
Create Test File

Let's create a test file so we can verify various aspects of the activity's view and behavior.

You can manually create the test file by clicking on the package in the robolectric_test module and adding a java class called CalculatorActivityTest.java.

However, I prefer to use Android Studio key shortcuts to generate a test in the correct folder.⁷

To do this, open the file CalculatorActivity.java and place your cursor on the name of the class. Use the Mac key combination + + to create a test.



If the configuration isn't set up just right, the IDE may want to put your test file in a generated folder or in the androidTest directory (if it is still around).

Note: If you are unable to successfully put it in the right directory with the shortcut, you can move it manually later by using Android Studio or the command line.

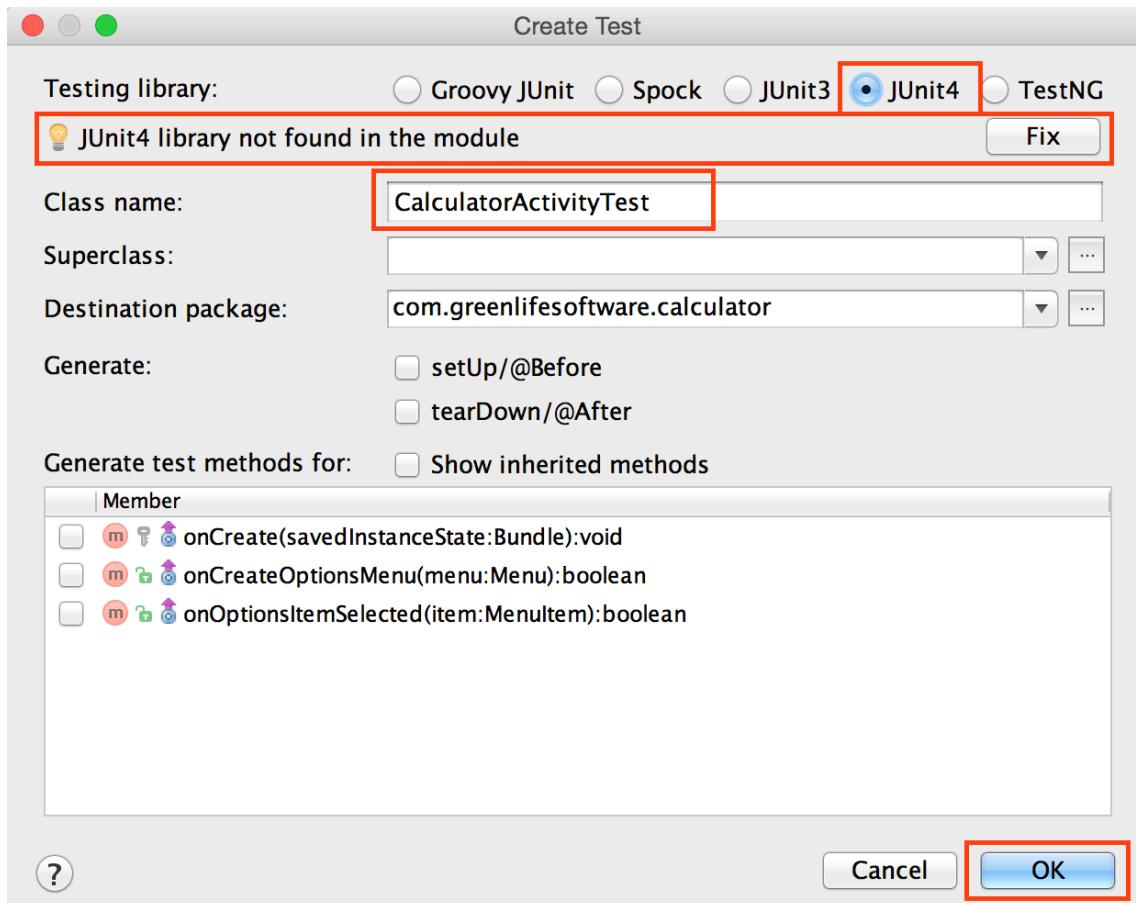
⁷Latislaw, C. Android Studio Cheatsheet.

There's not much to do in the dialog that the shortcut brings up.

Select `JUnit4` at the top of the dialog.

Confirm the test name is "CalculatorActivityTest."

Android Studio helpfully asks if you want add JUnit4 to your project. Click the `Fix` button to accept their help.



Click `OK` to be taken to your shiny new test file.

Test Runner

Android added “fangs” so that we can’t run Android code natively in the Java Virtual Machine (JVM). Robolectric helps us get around this restriction by using the test runner we created earlier.

```
@RunWith (RobolectricGradleTestRunner.class)
```

If we hadn’t used the test runner, we’d see this error when running the tests:

```
java.lang.RuntimeException: Stub!
    at android.content.Context.<init>(Context.java:4)
    at android.content.ContextWrapper.<init>(ContextWrapper.java:5)
    at android.view.ContextThemeWrapper.<init>(ContextThemeWrapper.java:5)
    at android.app.Activity.<init>(Activity.java:6)
    ...
...
```

You’ll see this error any time you forget to add the `@RunWith` annotation to a new test file. Now you know how to quickly fix it.

Set Up

Add a field `activity` so we can access our activity under test in all tests in this class.

Now use the `setUp()` function to populate the field.⁸

```
private CalculatorActivity activity;

@Before
public void setUp() throws Exception
{
    activity = Robolectric.buildActivity( CalculatorActivity.class )
        .create()
        .start()
        .resume()
        .get();
}
```

In `setUp()`, use `buildActivity()` and specify the class type to create the test object.

Then we use Robolectric's lifecycle management functions to cycle through the different phases of Android's activity lifecycle. The last chained function call, `get()`, returns a test instance that we assign to our field `activity`.

The calls to `create()`, `start()`, and `resume()` map to Android's activity lifecycle. This is covered in detail in the "Managing the Lifecycle" chapter of the Android Theory Book.⁹

⁸Normally with TDD you'd wait until the exact moment you need the code to write it. I prefer being practical over pedantic.

⁹TODO: Proper reference here.

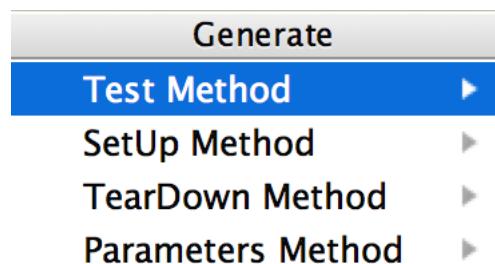
Add Test

The first test I write for each activity is always a sanity check. This helps verify the installation of the testing framework when first starting out and helps test the plumbing of subsequent activities.

You can write the test manually by typing the code below and importing `org.junit.Assert.assertNotNull`.

```
@Test  
public void shouldNotBeNull() throws Exception  
{  
    assertNotNull( activity );  
}
```

I prefer to generate the test with `⌘+N`.



After the generate options are shown, use `Enter` to select the top suggestion, `Test Method`.

Android Studio uses its test template¹⁰ to fill in the structure for you. It places the cursor in the name of the test (with **Name** selected) so you can change the default name easily.

Name the test “shouldNotNull.”

```
@Test  
public void shouldNotNull() throws Exception {  
}
```

When you hit  after naming the test, you will be taken to the first line inside the function.

Add the assertion `assertNotNull(activity);` and static import the library using the key combination  + .

```
@Test  
public void shouldNotNull() throws Exception  
{  
    assertNotNull( activity );  
}
```

 Create Method 'assertNotNull'
 Static Import Method...

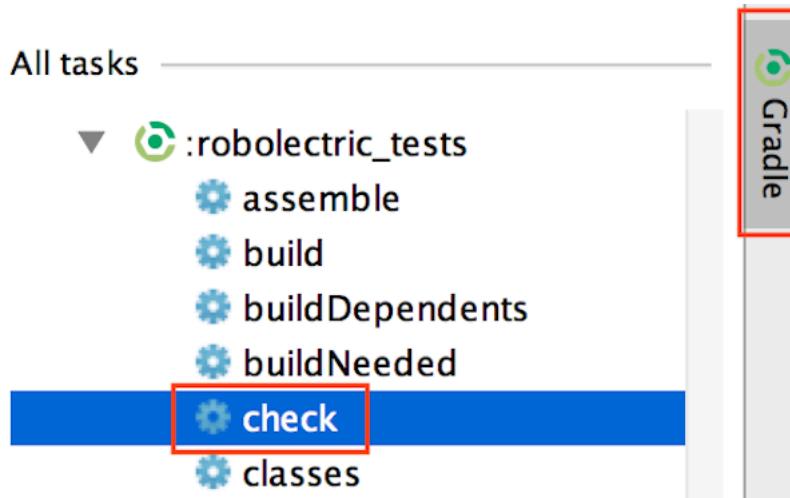
Now we’re ready to run the tests.

¹⁰This is configurable. I changed test names to begin with “should” instead of the default “test” so that my tests read more like a sentence.

3.5 Run Test

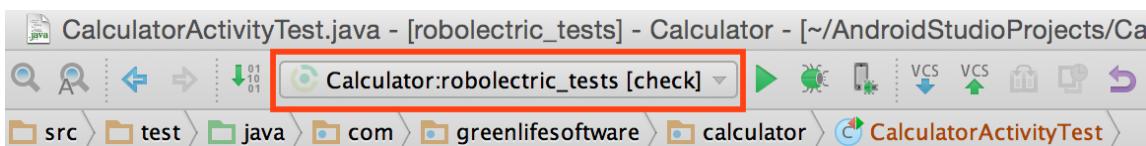
We can run the tests inside Android Studio. Open the Gradle tasks tab (right side of the IDE).

This window shows all the available Gradle tasks that you can run for the project or for a module. This listing is the same that you'd get by running `./gradlew tasks` on the command line.¹¹



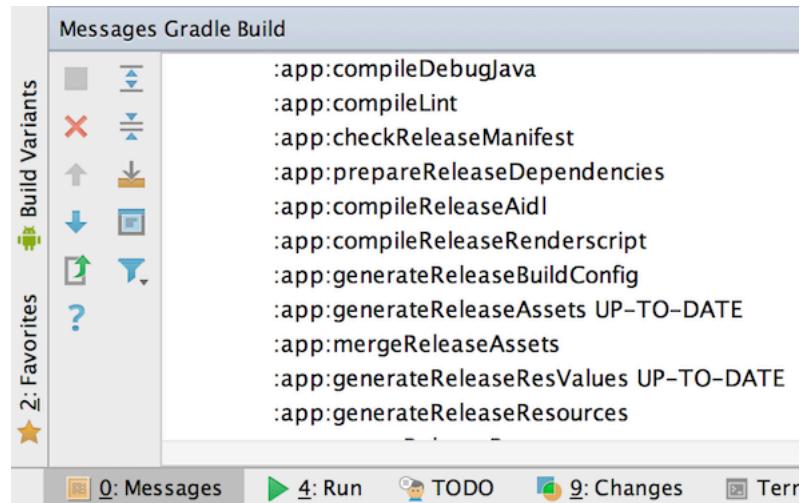
Find `check` under the `:robolectric_tests` module and double click to start the task.

This will start the tests and create a run configuration you can use in the future to replay this task.



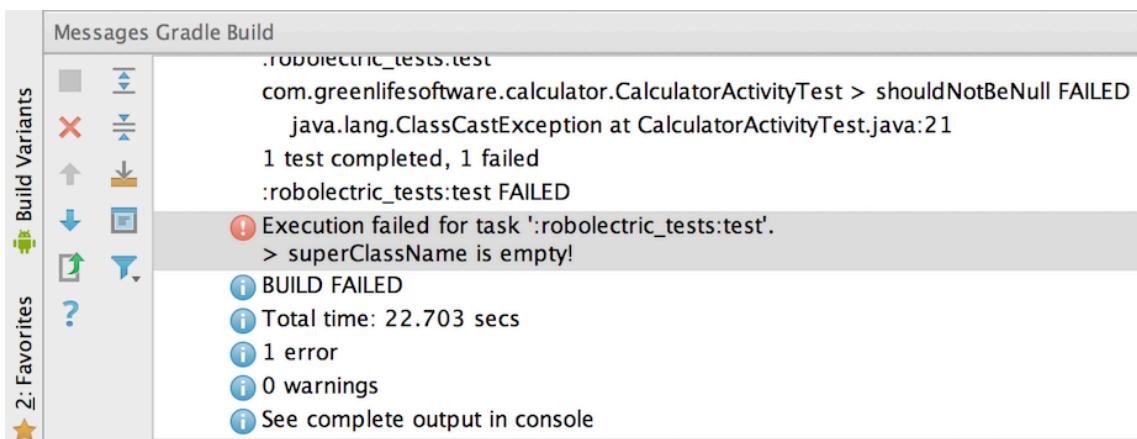
¹¹This arguably provides a better listing since they are grouped by type and have explanations. That said, if you hover over a task in the window, Android Studio will give a description of the task.

As Gradle goes through the [build cycle](#), messages are shown at the bottom left of the screen in the Messages tab.



If you add tasks and messages to the build file, then you will find them here. This is also where you look for build errors and test failures.

Speaking of build errors, we got one on the first run through.



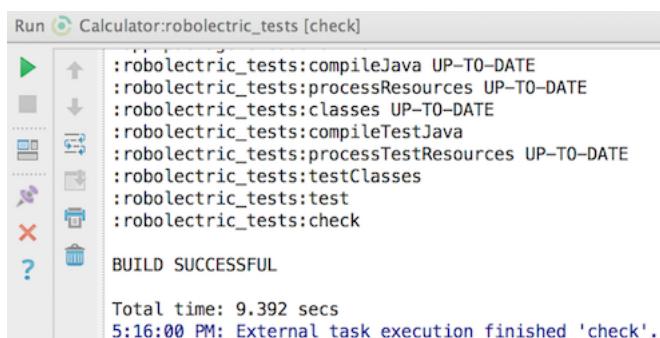
Gradle is having a hard time finding our test classes.

To fix this, we turn off class file scanning¹² and look for classes ending in “Test” in the file `build.gradle` in module `robolectric_tests`.

```
tasks.withType(Test)
{
    scanForTestClasses = false
    include "**/*Test.class"
}
```

After fixing the error, run the Gradle check command again.

Now our test passes.



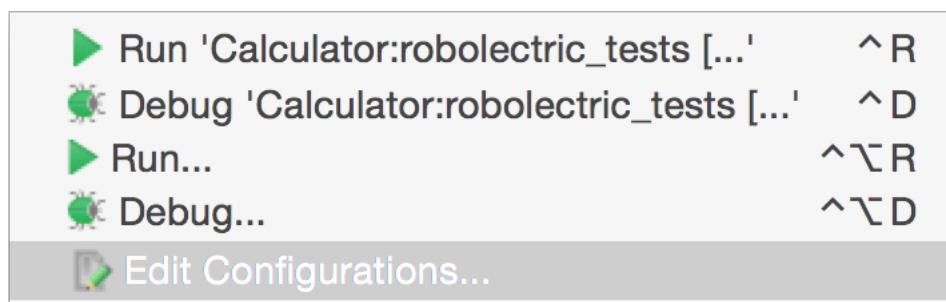
¹²Solution described in a [stackoverflow answer](#).

Run Configs

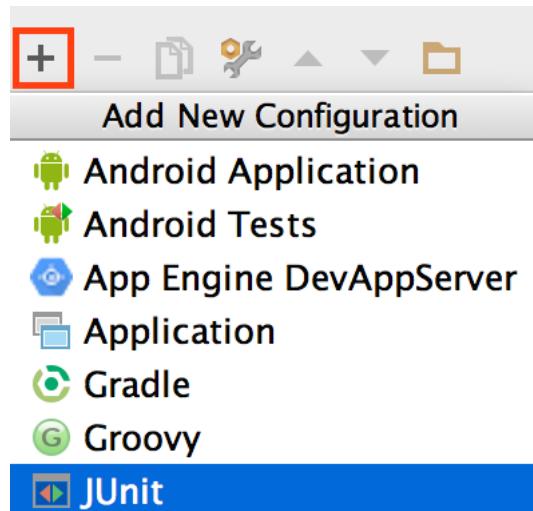
As I mentioned earlier, run configurations are created automatically when you click on a Gradle build task.¹³ However, you are not limited to these configurations and can define your own.¹⁴

We're going to create a run configuration for all of our unit tests.

First, choose **Edit Configurations ...** from the **Run** drop down menu.



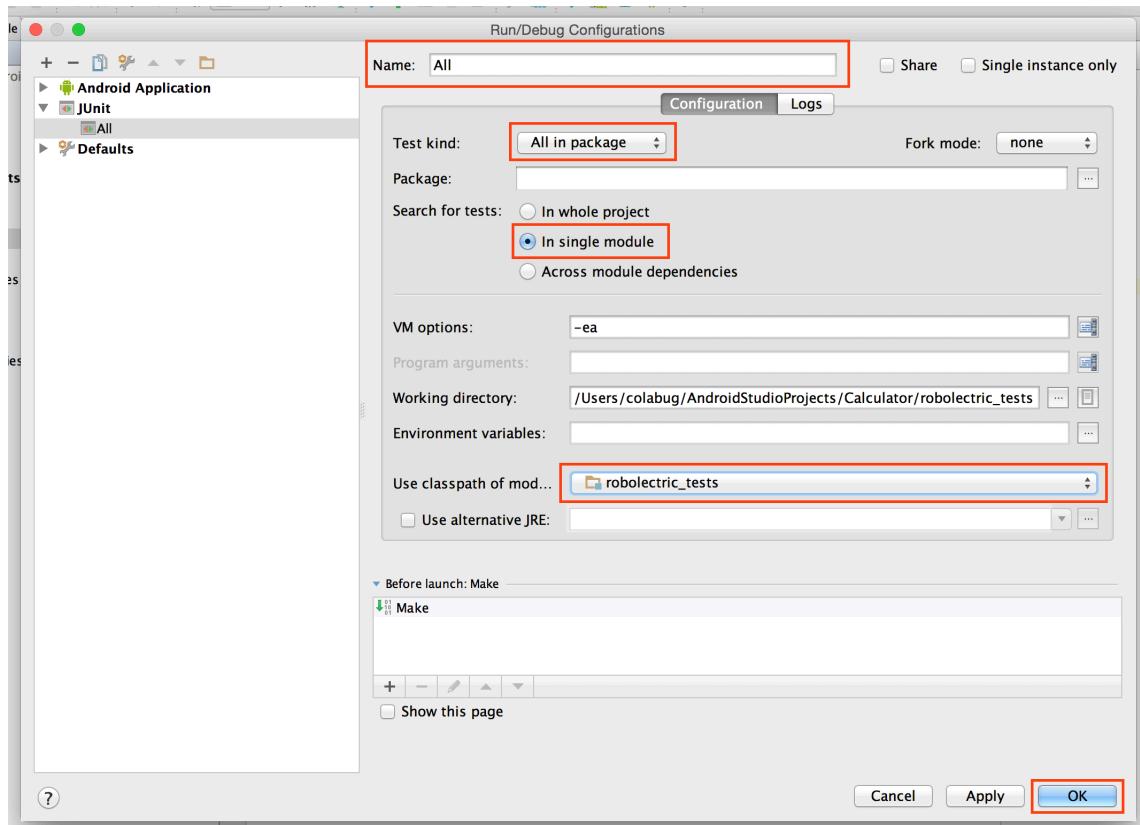
Then, hit the plus button and choose **JUnit**.



¹³They are also created if you right click on the test file name or a method name to run a test. You can delete these if you no longer need them by opening the **Edit Configurations ...** dialog.

¹⁴These configurations are useful for deploying the app to your device or emulator or running all tests before a commit.

We'll configure several pieces of information on this screen to run all tests at once.¹⁵



First, give your configuration a descriptive name such as “All Tests.”

Select **All in package**, then select **In single module**.

Enter “robolectric_tests” for **Use classpath of module**.

Click **OK** to apply your changes.

¹⁵For a more in depth treatment of the topic, check out Paul Blundell's [blog](#) on the subject.

3.6 Second Test

The default activity from the wizard has a string at the top of the screen. The string is “Hello world!” and is defined in the file: `app>src>main>res>values>strings.xml`

Create a test named `shouldHaveWelcomeString()` so we can verify that the welcome string is present and that the expected text is shown.¹⁶

```
@Test
public void shouldHaveWelcomeString() throws Exception
{
    TextView welcomeText = (TextView) activity.findViewById( R.id.welcome_text );

    assertViewIsVisible( welcomeText );

    assertThat( welcomeText.getText().toString(),
                equalTo( ResourceLocator.getString( R.string.hello_world ) ) );
}
```

The first line of the test fetches the view in question. We use our `activity` field to find the view defined with the `R.id.welcome_text` (we add the id on the next page).

The function `findViewById()` returns a `View` object and we cast it to a `TextView` so that we can use the function `getText()` later in the test.

Now that we have access to the view element, we verify that the view is non-`null` and visible. We define the function `assertViewIsVisible()` in the next section.

The second assertion ensures that the string shown in the view matches the string that we expect. We define the utility function `ResourceLocator.getString()` shortly.

¹⁶In TDD, all code that exists need a test and code without tests should die.

Since the view does not come with an id automatically, we'll need to add one.

To locate the `.xml` file that contains the view, we look at the activity's `setContentView()` to see which file it specifies with the `R.layout` parameter. In this case, the name is `activity_calculator`.

```
@Override  
protected void onCreate( Bundle savedInstanceState )  
{  
    super.onCreate( savedInstanceState );  
    setContentView( R.layout.activity_calculator );  
}
```

Open the `activity_calculator.xml` file and add an `id` with the name `welcome_text` to the `TextView`.

```
<TextView  
    android:id="@+id/welcome_text"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/hello_world" />
```

There are several ways to open this file. You can hold `⌘` and click on the word `activity_calculator`, you can use the key combination `⌘ + B` while on the word `activity_calculator`, you can use the key combination `⌘ + O` from anywhere and type “activity_calculator”, or you can navigate there with the project browser and mouse.

Visibility

Visibility and non-**null**-ness is a common test for view elements in your application. Create a file `Assert.java` in the `support` package of your `robolectric_tests` module.

```
public class Assert
{
    public static void assertViewIsVisible( View view )
    {
        assertNotNull( view );
        assertThat( view.getVisibility(), equalTo( View.VISIBLE ) );
    }
}
```

Create `ResourceLocator.java`

Another common verification is that resources specified for a view are what we expect. Here we add easy access to strings and drawables used in the application. This utility function takes the id and fetches the appropriate resource. We cover resources and resource qualifiers in depth in the “Resource System” chapter of the Android Theory Book.

Strings that are defined in xml are accessed through an id in the form `R.string.name`. Drawables that are saved in one of the `drawable` folders are accessed with `R.drawable.name`.

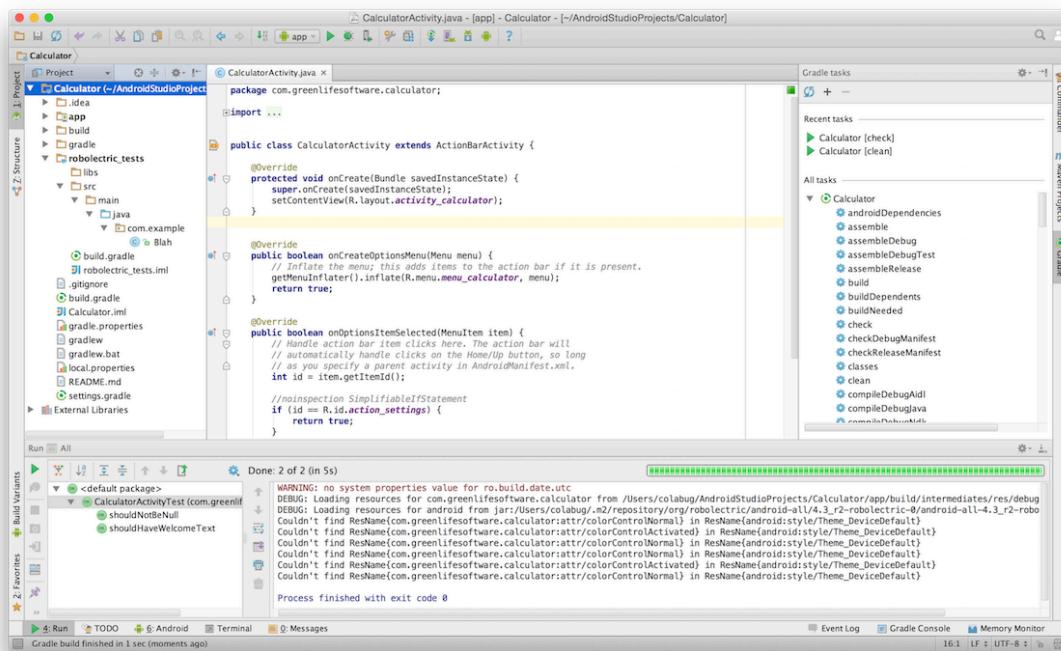
```
public class ResourceLocator
{
    public static String getString( int stringId )
    {
        return Robolectric.application
            .getString( stringId );
    }

    public static Drawable getDrawable( int drawableId )
    {
        return Robolectric.application
            .getResources()
            .getDrawable( drawableId );
    }
}
```

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.

Code Listing

This code is what we added step by step and listed here for reference. You can obtain the latest version of the code in the example project.¹⁸

Application Code

These files are located in the  app module.

CalculatorActivity.java

 app>src>main>java>com>greenlifesoftware>calculator

```
package com.greenlifesoftware.calculator;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

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

    // Removed unrelated option menu code generated by the wizard
}
```

¹⁸TODO: Insert link

activity_calculator.xml

app>src>main>res>layout

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".CalculatorActivity">

    <TextView
        android:id="@+id/welcome_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

Test Module

These files are located in the `robolectric_tests` module.

`build.gradle`

`robolectric_tests`

```
apply plugin: 'java'

dependencies {
    def androidModule = project(':app')
    compile androidModule

    testCompile androidModule.android.applicationVariants.toList().first()
                .javaCompile.classpath
    testCompile androidModule.android.applicationVariants.toList().first()
                .javaCompile.outputs.files
    testCompile files(androidModule.plugins
                    .findPlugin("com.android.application")
                    .getBootClasspath())

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

tasks.withType(Test) {
    scanForTestClasses = false
    include "**/*Test.class"
}
```

CalculatorActivityTest.java

robolectric_tests▶src▶test▶java▶com▶greenlifesoftware
▶calculator

```
package com.greenlifesoftware.calculator;

import android.widget.TextView;

import com.greenlifesoftware.calculator.support.ResourceLocator;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.robolectric.Robolectric;

import static com.greenlifesoftware.calculator.support.Assert.assertViewIsVisible;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertThat;

@RunWith(RobolectricGradleTestRunner.class)

public class CalculatorActivityTest
{

    private CalculatorActivity activity;

    @Before
    public void setUp() throws Exception
    {
        activity = Robolectric.buildActivity( CalculatorActivity.class )
            .create().start().resume()
            .get();
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( activity );
    }

    @Test
    public void shouldHaveWelcomeString() throws Exception
    {
        TextView welcomeText = (TextView) activity.findViewById(R.id.welcome_text);
        assertViewIsVisible( welcomeText );
        assertThat( welcomeText.getText().toString(),
                    equalTo( ResourceLocator.getString( R.string.hello_world ) ) );
    }
}
```

📁 RobolectricGradleTestRunner.java

📁 robolectric_tests▶src▶test▶java▶com▶greenlifesoftware
▶calculator▶support

```
package com.greenlifesoftware.calculator.support;

import org.junit.runners.model.InitializationError;
import org.robolectric.AndroidManifest;
import org.robolectric.RobolectricTestRunner;
import org.robolectric.annotation.Config;
import org.robolectric.res.Fs;

import java.io.File;
import java.io.IOException;

public class RobolectricGradleTestRunner extends RobolectricTestRunner
{
    private static final String EMPTY_STRING = "";
    private static final String PATH_SEPARATOR = "/";
    private static final String CURRENT_DIRECTORY = ".";

    // Project modules
    private static final String APP_MODULE = "app";
    private static final String ROBOLECTRIC_MODULE = "robolectric_tests";

    // Max SDK version
    private static final int MAX_SDK_SUPPORTED_BY_ROBOLECTRIC = 18;

    // Project locations
    private static final String MANIFEST_PATH = "src/main/AndroidManifest.xml";
    private static final String RES_PATH = "build/intermediates/res/debug/";

    // Path configuration
    private static final String PROJECT_DIR = getProjectDirectory();
    private static final String FINAL_MANIFEST_PATH = PROJECT_DIR + MANIFEST_PATH;
    private static final String FINAL_RES_PATH = PROJECT_DIR + RES_PATH;
```

```

public RobolectricGradleTestRunner( final Class<?> testClass )
    throws InitializationError
{
    super( testClass );
}

@Override
public AndroidManifest getAppManifest( Config config )
{
    return new AndroidManifest( Fs.fileFromPath( FINAL_MANIFEST_PATH ),
                               Fs.fileFromPath( FINAL_RES_PATH ) )
    {
        @Override
        public int getTargetSdkVersion()
        {
            return MAX_SDK_SUPPORTED_BY_ROBOLECTRIC;
        }
    };
}

private static String getProjectDirectory()
{
    String path = EMPTY_STRING;
    try
    {
        File file = new File( CURRENT_DIRECTORY );
        path = file.getCanonicalPath();

        // Test Module
        path = path.replace( ROBOLECTRIC_MODULE, EMPTY_STRING );

        // Application Module
        path = path.replace( APP_MODULE, EMPTY_STRING );
        path = path + PATH_SEPARATOR + APP_MODULE + PATH_SEPARATOR;
    }
    catch ( IOException ignored )
    {
    }

    return path;
}
}

```

❑ Assert.java

❑ robolectric_tests▶src▶test▶java▶com▶greenlifesoftware
▶calculator▶support

```
package com.greenlifesoftware.calculator.support;

import android.view.View;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertThat;

public class Assert
{
    public static void assertViewIsVisible( View view )
    {
        assertNotNull( view );
        assertThat( view.getVisibility(), equalTo( View.VISIBLE ) );
    }
}
```

❑ ResourceLocator.java

❑ robolectric_tests▶src▶test▶java▶com▶greenlifesoftware
▶calculator▶support

```
package com.greenlifesoftware.calculator.support;

import android.graphics.drawable.Drawable;

import org.robolectric.Robolectric;

public class ResourceLocator
{
    public static String getString( int stringId )
    {
        return Robolectric.application
            .getString( stringId );
    }

    public static Drawable getDrawable( int drawableId )
    {
        return Robolectric.application
            .getResources()
            .getDrawable( drawableId );
    }
}
```

Chapter 4

Building the Display

In this chapter, we'll create the display to show calculated results. To do this, we create a fragment and then host it in the `CalculatorActivity`.



Before we write code, we'll write a test. Your goal is to build the next smallest piece of functionality. We write a test that forces us to move forward,¹ instead of writing a lot of code and retrofitting it with tests later.

We will write code that doesn't compile and use the IDE to generate the code we need. It's much faster than writing out everything the old-fashioned way.

Finally, after all the views are in place, we'll layout and style the display. Remember, before moving on to new things, tests should pass for the current component. Also, before each commit, all tests for the project should pass.

¹We'll be focusing on ensuring that activity and display elements are non-`null`, visible, and that proper resources are used.

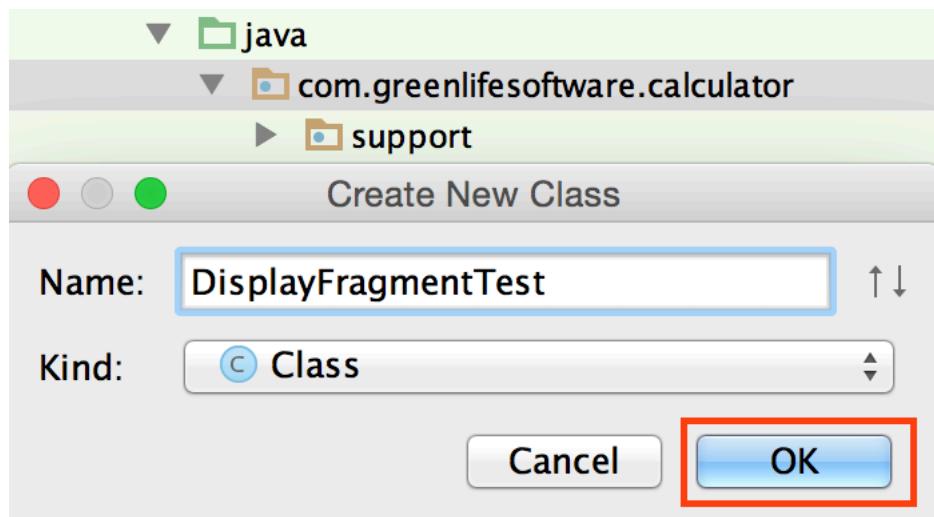
4.1 Create Test File

First, create a file called `DisplayFragmentTest` in the module `robolectric_tests`.

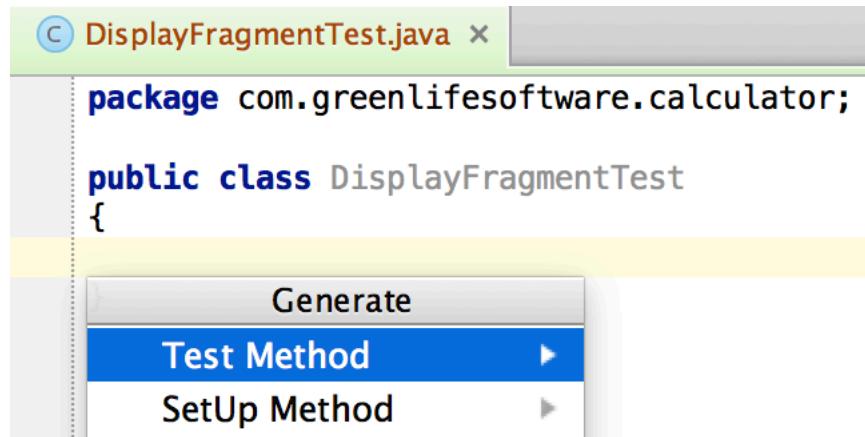
You can do this manually by creating a class in the folder `src/test/java/com/greenlifesoftware/calculator/`.

However, my preferred method is to right-click on the package name `com.greenlifesoftware.calculator` and select “New” -> “Java Class”. Then we simply name the test `DisplayFragmentTest`.

Click “OK” to create the file.

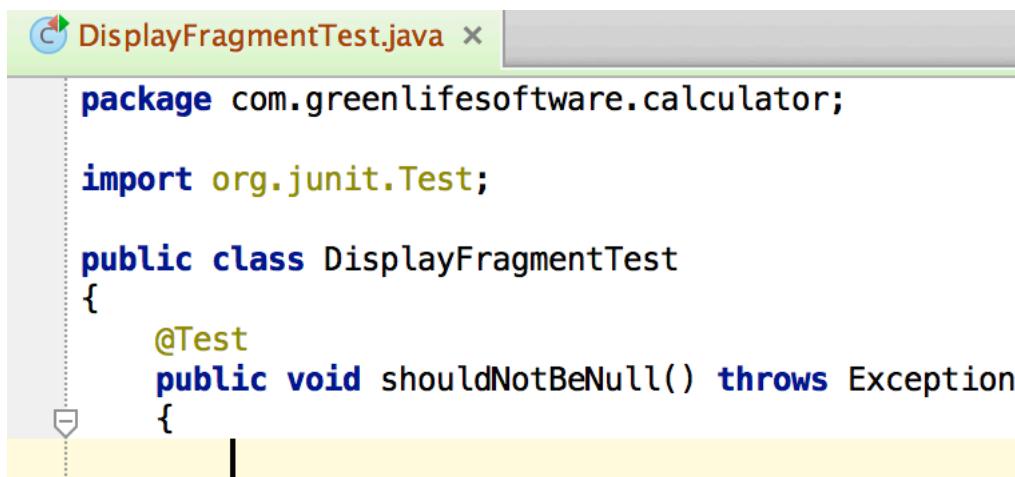


To add your first test, you can either type out the boilerplate manually or use the key combination **Command N** to generate the test for you.



When generating, select the first option, “Test Method,” and then “JUnit4” as the framework, if requested. Name your test `shouldNotBeNull()`.

When you accept the test name by hitting enter, you’ll be taken to the first line of your new function.



First we'll add the assertion that our test class exists. Then we'll use the IDE to will it into existence.

Add `assertNotNull(new DisplayFragment())`; to your test and you'll end up with two build errors.

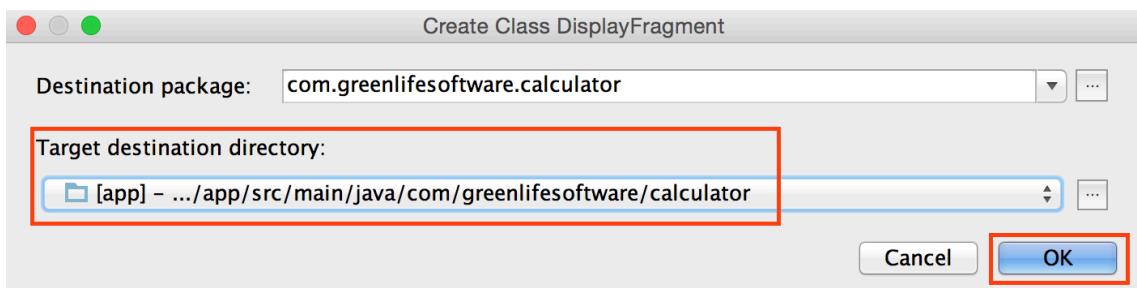
To fix the first one for `assertNotNull`, use the IDE's shortcut **Option Enter** to import `org.junit.Assert.assertNotNull`.

To fix the second error, place your cursor on the word `DisplayFragment` (in red) and use the shortcut **Option Enter** to generate a class.

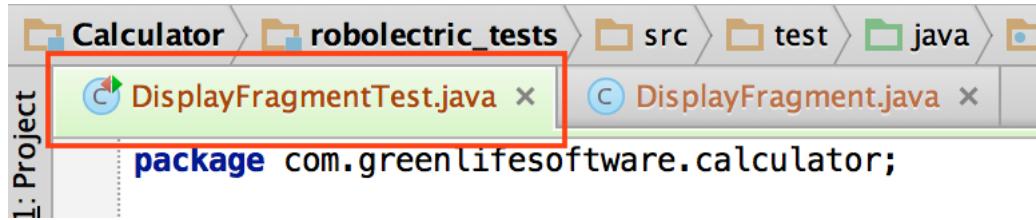
Select “Create Class ‘DisplayFragment’” from the pop-up.



Ensure that the `app` module is selected in the next dialog. The default selection is `robolectric_tests`, which isn't what we want.



Now run your test by right clicking on the file tab. This will create a temporary run configuration you can use while building out this class.



You can also run the class by using the “All Tests” configuration (created in [Run Configs](#)).

The test should pass. Commit your work.

4.2 Start Test Fragment

Extend Fragment

Open the file we just generated by using **Command O** and typing the name of the file (**DisplayFragment**), by clicking on the file in the file browser, or by the shortcut command **Command Shift T** when in the test file.²

First, this class should extend **Fragment**.³

Second, we'll add an empty constructor.

Note: The [documentation](#) suggest adding this constructor any time you create a **Fragment** subclass, you should add this constructor.

You can manually type the method signature or use the shortcut **Control O** to list override-able methods for **Fragment** and the constructor will be at the top of the list.

The class should look like this:

```
import android.support.v4.app.Fragment;

public class DisplayFragment extends Fragment
{
    public DisplayFragment()
    {
    }
}
```

The tests still pass.

²You can use this handy shortcut to switch back and forth between the test and class files.

³We'll be using **v4** support fragments throughout the book instead of platform fragments.

Start Fragment

To start testing view attributes, we'll need to use Robolectric's `startFragment()` API to create a testable version of the fragment.

When we specified our test earlier, we created our `DisplayFragment` inline in the call to `assertNotNull()`.

```
assertNotNull( new DisplayFragment() );
```

We need to refactor our test to create a variable for the fragment so we can also use it in the `startFragment()` call.

Note: The perfect time to refactor is when you're in a green state.
You can ensure your tests still pass after each step of your refactor.
You have a safety net and can easily rollback breaking changes.
On the other hand, if you refactored in the red state, you wouldn't know what broke the tests or how to fix it.

You could manually create a variable and use in the `assertNotNull()` call, but I prefer to use the IDE's tools to do it for me.

To abstract out a variable, highlight the call to the fragment constructor and use the key combination **Command Option V**.⁴ The shortcut automatically updates the call to `assertNotNull()` for you.

⁴One of the key combinations I use the most is progressive scope selection. This works by slowly expanding the amount of code that is selected. If your cursor is somewhere inside the parenthesis in the call to `assertNotNull()`, you can use **Option UP** twice to select the text `new DisplayFragment()`. Then you use the key combination to create a local variable.

After adding the call to `startFragment()`, your test should look like this:

```
DisplayFragment fragment = new DisplayFragment();
startFragment( fragment );
assertNotNull( fragment );
```

When you run the test this time, it fails with the dreaded “Stub!” exception because we started using Android APIs (by extending `Fragment` in the `DisplayFragment` file).

```
Caused by: java.lang.RuntimeException: Stub!
    at android.content.Context.<init>(Context.java:4)
    at android.content.ContextWrapper.<init>(ContextWrapper.java:5)
    at android.view.ContextThemeWrapper.<init>(ContextThemeWrapper.java:5)
    at android.app.Activity.<init>(Activity.java:6)
    at android.support.v4.app.FragmentActivity.<init>(FragmentActivity.java:75)
    ...
...
```

Test Runner

To fix the “Stub!” error, use the custom test runner created in the [Custom Test Runner](#) section.

Add the `@RunWith` annotation to the top of the test class (before the class definition).

```
@RunWith(RobolectricGradleTestRunner.class)
public class DisplayFragmentTest { ... }
```

The tests should pass again.

Let’s commit since we’re back in the green state.

4.3 Display View

Create Test

Now we're ready to start testing the calculator display view.

Create a test named `shouldHaveDisplay()` that will assert that our view is visible and non-`null`.

We'll need to create and start a fragment again in this test to get access to the view.

```
@Test
public void shouldHaveDisplay() throws Exception
{
    DisplayFragment fragment = new DisplayFragment();
    startFragment( fragment );
    assertViewIsVisible( fragment.getView()
        .findViewById( R.id.calculator_display ) );
}
```

Note: We'll refactor this code duplication later in the [Refactor: setUp\(\)](#) section.

Once we have a reference to the fragment, we use `getView()` and `findViewById()` to fetch the actual views.

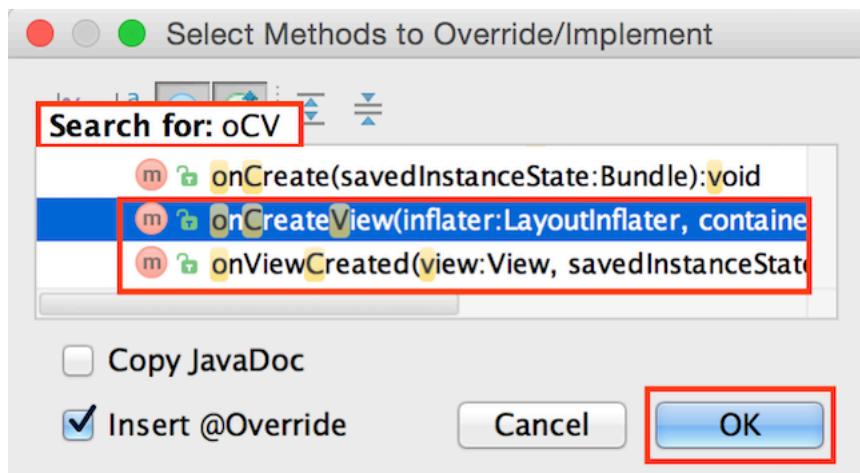
You'll notice a compilation error because we are trying to look up an id `calculator_display`, but it doesn't exist yet. We'll use the IDE to generate this view and id in the following sections.

Inflate View

To construct the display view, we'll override the lifecycle method `onCreateView()`.

Use the shortcut **Control O** (mentioned in the last section) to get a list of the methods you can override.

You can scroll through the list or start typing letters into the dialog to search.⁵



Note: There are two very similarly named functions to choose from when overriding: `onViewCreated()` and `onCreateView()`. According to the [docs](#), `onCreateView()` is immediately followed by `onViewCreated()`. Despite the past tense, I find this terribly confusing.

⁵You can use the camel casing shortcut shown by the search term at the top. This camel casing shortcut applies can be used in many places in the IDE such as opening files and during code completion.

Inside `onCreateView()`, we construct our view from the layout file, the layout inflater `inflater`, and the view `container`.

```
@Override  
public View onCreateView( LayoutInflater inflater,  
                          ViewGroup container,  
                          Bundle savedInstanceState )  
{  
    View layout = inflater.inflate( R.layout.fragment_display,  
                                  container,  
                                  false );  
    return layout;  
}
```

The layout file (generated in the next section, [XML](#)) is displayed here as `R.layout.fragment_display`.

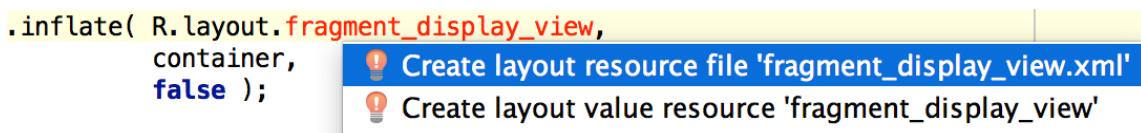
Note: The build process creates a file called `R.java` behind the scenes that gives us access to resources in Java-land. The format to access these resources is `R.type.name`. The xml file will be located in the `app/src/main/res/layout/` directory.

After the view is inflated, you have access to the other views specified in the layout file. You use the variable `layout` to get references to view components and add user interact-able elements (covered in depth in [Chapter 7](#)).

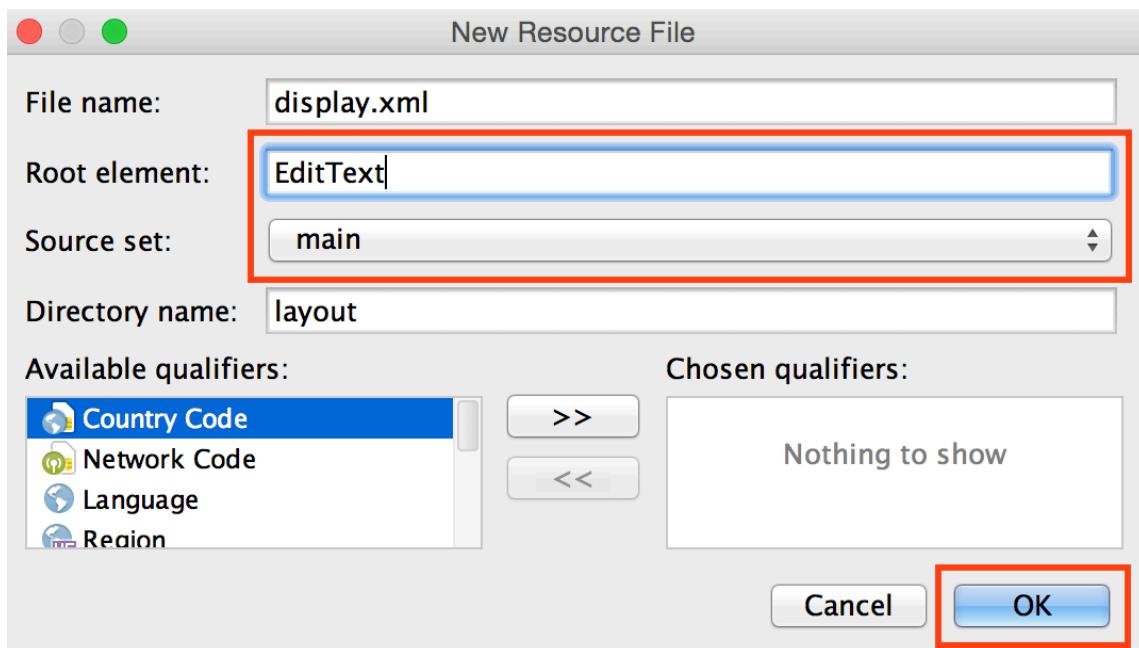
After we're done configuring the view, we return it at the end of the function.

XML

The XML view doesn't exist, so we'll generate it using IDE shortcuts. Put your cursor on the word `R.layout.fragment_display` (in red) and use the shortcut **Option Enter** to bring up a dialog. Choose the top option, “Create layout resource file ‘fragment_display.xml’.”



When the “New Resource File” dialog pops up, enter `EditText` as the “Root element.” We’ll leave the “Source set” as `main`⁶. Click “OK.”



⁶For more information about build types and flavors, check out Gradle’s documentation.

This creates an XML file for you.

```
<?xml version="1.0" encoding="utf-8"?>  
  
<EditText  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
</EditText>
```

The last step to get the tests to pass is to add an id attribute to the `EditText` view with the same name that we used in the test, `calculator_display`.

```
    android:id="@+id/calculator_display"
```

Run the test and it should pass.

Refactor: `setUp()`

In our test file, we duplicated code - a violation of the [DRY principle](#). When using TDD, we do the simplest things to get the tests to pass and clean up after ourselves when we are in a green state.

We're in a green state now, so let's improve the code.

Open the `DisplayFragmentTest` file and navigate to the first test, `shouldNotBeNull()`.

One thing to notice is that we created the same fragment variable in each test. If we promoted that variable to a field and populated in a `setUp()` function, both tests would have access.

You can create this field manually, or put your cursor on the `fragment` local variable and use the key combination **Command Option F** to promote it to a field.

You have the choice of where to initialize the variable. Choose “`setUp`” here so it will generate the function for you and clean up the initialization in your test.

```
private DisplayFragment fragment;
pr
@T
pu
{
    DisplayFragment fragment = new DisplayFragment();
    startFragment( fragment );
    assertNotNull( fragment );
}
```

Note: Unfortunately you won't be able to replace both instances of `fragment` at once.

After using the command, the first test uses the new `fragment` field, which is populated in `setUp()`. Let's delete the initialization of `fragment` from `shouldHaveDisplay()` as well.

Our tests also repeat the call to `startFragment()`. Let's move one of these calls from a test to the `setUp()`. Delete the line from the other test.

The tests should pass after this refactor.

Refactor: Factory Pattern

It's **bad practice** to call the constructor directly. Instead, we'll use the factory pattern to generate one for us. This also gives us the ability to pass arguments when constructing a **DisplayFragment**.

In the **setUp**, function, **fragment** initialization should use **DisplayFragment.newInstance()** instead of the constructor.

```
@Before  
public void setUp() throws Exception  
{  
    fragment = DisplayFragment.newInstance();  
    startFragment( fragment );  
}
```

Use the shortcut **Option Enter** when your cursor is on **newInstance** (shown in red) to automatically generate the function for you in the **DisplayFragment** class.

The IDE will open to **DisplayFragment** for you. Simply return a new instance of the fragment.

```
public static DisplayFragment newInstance()  
{  
    return new DisplayFragment();  
}
```

Post-refactor, your **DisplayFragmentTest** class looks like:

```
@RunWith(RobolectricGradleTestRunner.class)
public class DisplayFragmentTest
{
    private DisplayFragment fragment;

    @Before
    public void setUp() throws Exception
    {
        fragment = DisplayFragment.newInstance();
        startFragment( fragment );
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( fragment );
    }

    @Test
    public void shouldHaveDisplay() throws Exception
    {
        assertViewIsVisible( fragment.getView()
                            .findViewById( R.id.calculator_display ) );
    }
}
```

Run the tests. Commit when they pass.

4.4 Default Display

Test Default Display

Create a test called `shouldHaveDefaultDisplay()` to verify our initial view state.

We already fetched the display view in the `shouldHaveDisplay()` test. Let's promote this to a field which is initialized in `setUp()` using **Command Option F** so we can use it in all tests.

Note: When we first fetched the view, we stored it as a `View` object, which is what `findViewById()` returns by default.

We are using functionality that's specific to `EditText` in our test, so we'll need to change our field type and cast the result of `findViewById` to an `EditText`.

Our setup and field now look like this:

```
private EditText display;

@Before
public void setUp() throws Exception
{
    fragment = DisplayFragment.newInstance();
    startFragment( fragment );

    display = (EditText) fragment.getView()
        .findViewById( R.id.calculator_display );
}
```

Now we're ready to add the assertion to our test.

```
@Test  
public void shouldHaveDefaultDisplay() throws Exception  
{  
    assertThat( display.getText().toString(),  
                equalTo( getString( R.string.DEFAULT_DISPLAY ) ) );  
}
```

We fetch the characters shown in the display with `getText()` (cast to a string with `toString()`), fetch the string resource using the `ResourceLocator` utility (created in the [Create ResourceLocator.java](#) section), and compare them using `equalTo()`.

The string we want doesn't exist yet, so add a string⁷ to `strings.xml`.⁸

```
<string name="DEFAULT_DISPLAY">Display</string>
```

When we run the test, we get an assertion failure.

```
java.lang.AssertionError: Expected: "Display" but: was ""
```

This happens because we haven't set the text in the view's xml (default is an empty string). To fix the test failure, add the string to the `EditText` in `fragment_display.xml`.

```
<EditText  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:text="@string/DEFAULT_DISPLAY">  
</EditText>
```

⁷Non-blank for testing purposes.

⁸There's no fancy way to generate the string from the test file, but you *can* automatically generate it from the XML file using a similar process outlined in the [Add Color File](#) section.

4.5 Show Display

So far we've been creating our `DisplayFragment` in isolation. We couldn't actually see what we've created because our fragment wasn't attached to an activity.

The activity that is launched when we start the application is `CalculatorActivity`.⁹ We'll add the fragment to its view.

Test Presence

Add a test named `shouldHaveDisplayFragment()` to `CalculatorActivityTest`.

```
@Test
public void shouldHaveDisplayFragment() throws Exception
{
    assertNotNull( activity.getSupportFragmentManager()
                    .findFragmentById( R.id.display_fragment ) );
}
```

You can't generate the fragment xml automatically.

Add Fragment

Add a fragment to `activity_calculator.xml`.¹⁰

```
<fragment
    android:id="@+id/display_fragment"
    android:name="com.greenlifesoftware.calculator.DisplayFragment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

⁹For more information on how to create a new project using Android Studio, check out my book “Hello, Android Studio.”

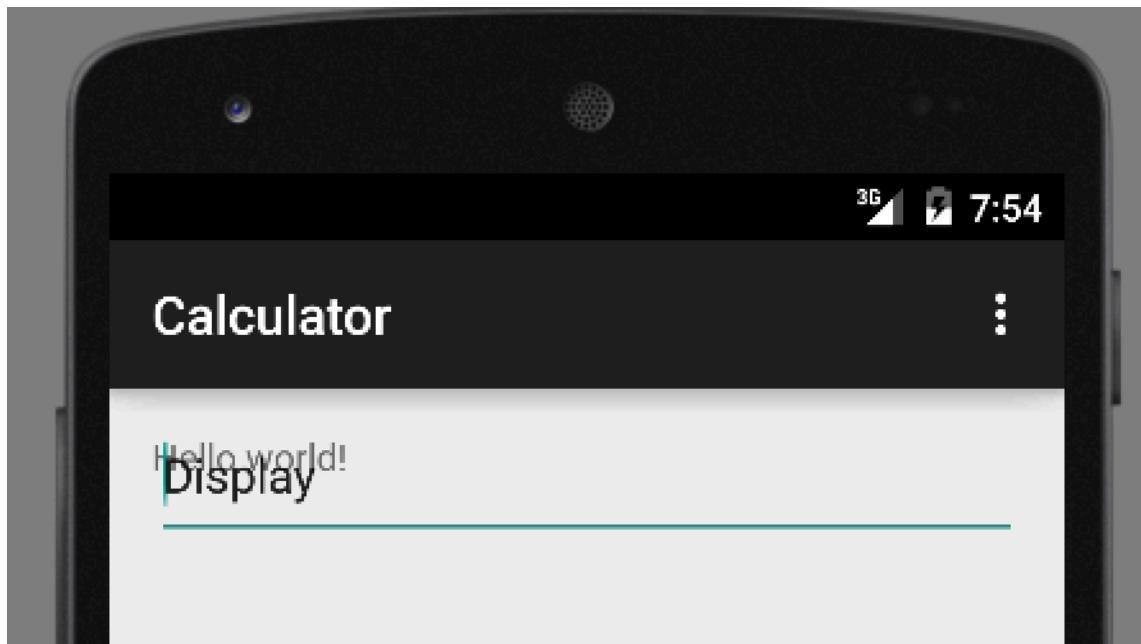
¹⁰To determine which layout file is associated with `CalculatorActivity`, look at the resource `R.layout.activity_calculator` specified in `setContentView()` in the `onCreate()` lifecycle method.

Use the same id you specified in the test. Specify the fully qualified path for the `name` attribute (code completion possible). Finally, make the view fill the width of the screen and only take up as much space as necessary for the height.

The tests should pass.

View App

Now that the tests are passing, let's view the app.



You should see your display fragment overlapping the welcome text (comes with a new project if you use the wizard).

The view doesn't look great, let's add some style!

4.6 Display Styling

Let's add some styling to the view.

Note: We're not going to add tests for this section since it'd be overkill to confirm view elements that are likely to change often.

Add the following attributes to the `EditText` in our `DisplayFragment`. Each attribute affects the appearance or the behavior of the view. All available attributes for `EditText` are specified in the [documentation](#).

```
<?xml version="1.0" encoding="utf-8"?>
<EditText
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/calculator_display"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:paddingTop="20dp"
    android:paddingBottom="20dp"
    android:gravity="center"
    android:textSize="40sp"
    android:textStyle="bold"
    android:textColor="@color/nearly_black"
    android:text="@string/DEFAULT_DISPLAY"
    android:editable="false"
    android:maxLength="10"
    android:inputType="none"
    android:focusable="false"
    android:focusableInTouchMode="false"
    android:background="@android:color/white">
</EditText>
```

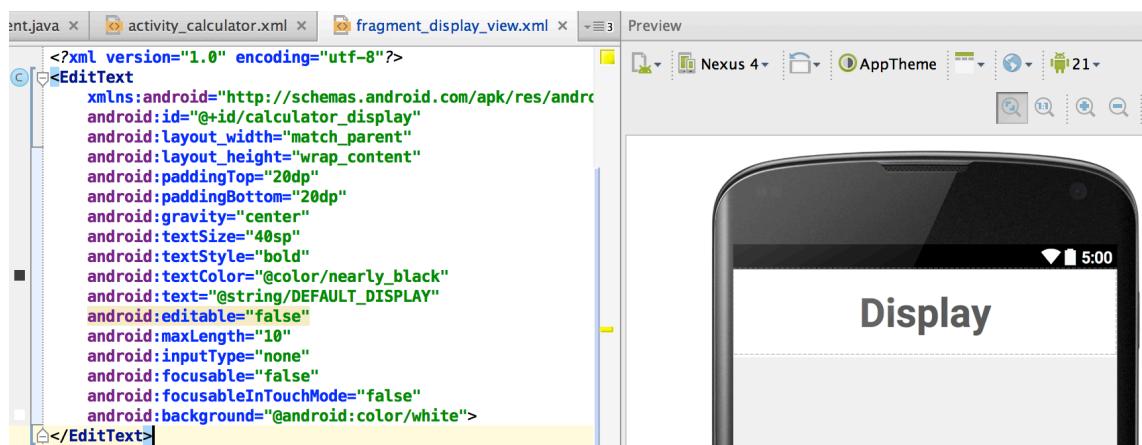
The padding gives the view breathing room. The text styling make the text a bit bigger, centered, bolder, and a specific color. We make the field non-editable, non-focusable, and set the input type to none so that it only shows the text we tell it to display. We limit the length of the display to 10 characters. Finally we set the background color.¹¹

¹¹Ordinarily we wouldn't hard-code all these values directly into the view. We'll cover abstracting dimensions and styles in the Resource System chapter of *Hello, Android Studio*. This book also covers the difference between `dp` and `sp`.

For the **background** color, I'm using a builtin Android resource, but have specified a custom color for the **textColor**. We'll define our colors in the next section, [Add Color File](#).

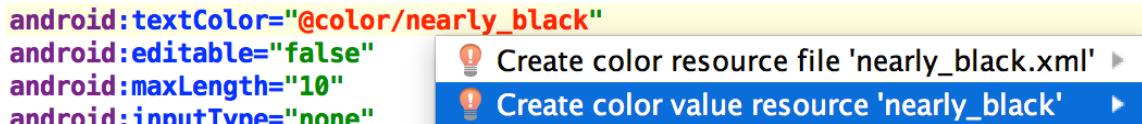
```
    android:textColor="@color/nearly_black"
    android:background="@android:color/white"
```

Pro tip: The preview window is helpful when prettying up the view.

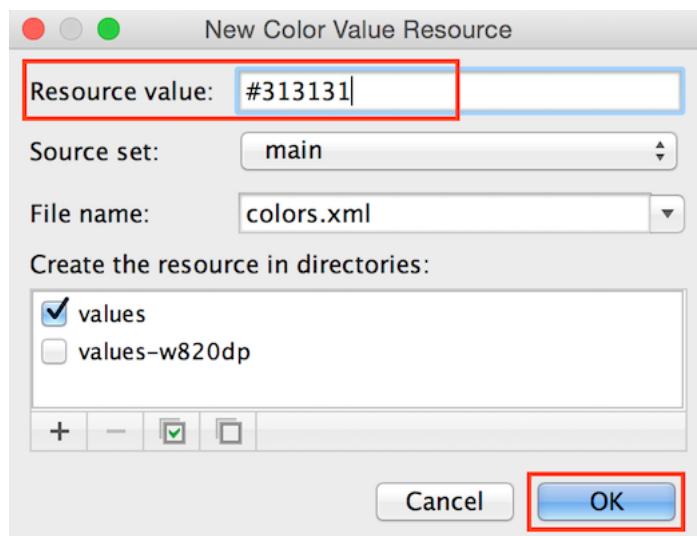


Add Color File

We can use the IDE to generate the color file for us. Use the key combination **Option Enter** to bring up the window. Select “Create color value resource ‘nearly_black’.”



In the “New Color Value Resource” dialog, enter the **RGB value #313131** in the “Resource value” section. Click “OK.”

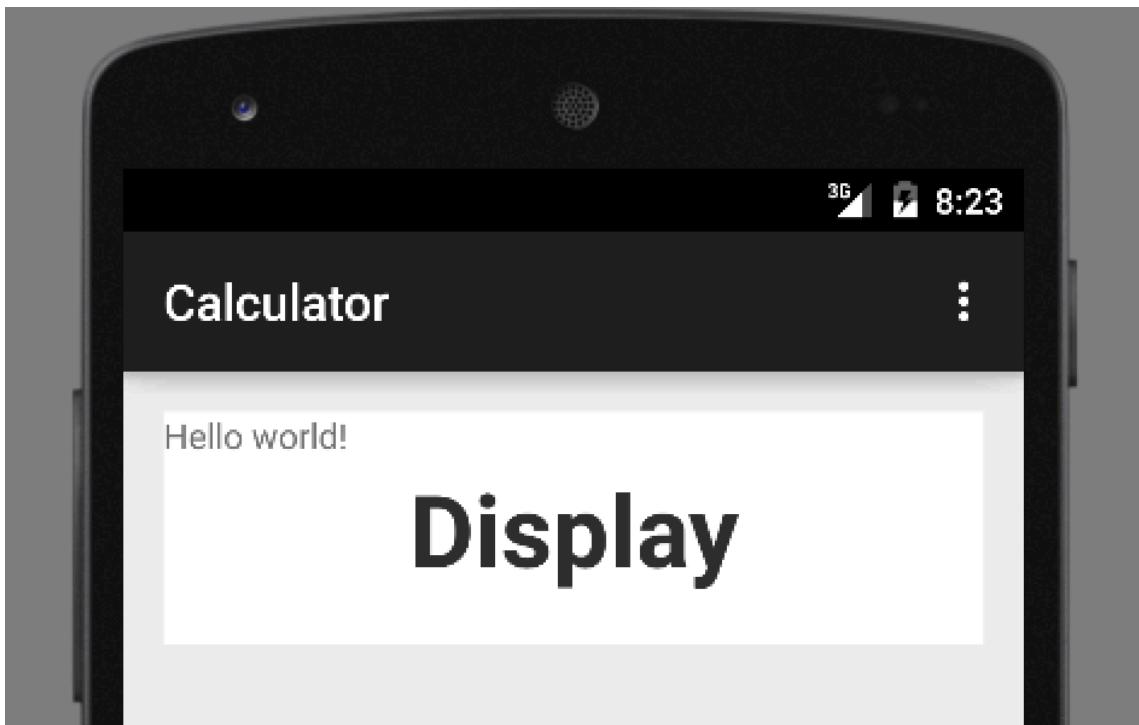


This creates a color file at **app/main/res/values/colors.xml** and adds our color to it:

```
<resources>
    <color name="nearly_black">#313131</color>
</resources>
```

Styled View

Run the app to see the styling in action.



Much better! However, there's still the pesky welcome text.¹²

Clean Up

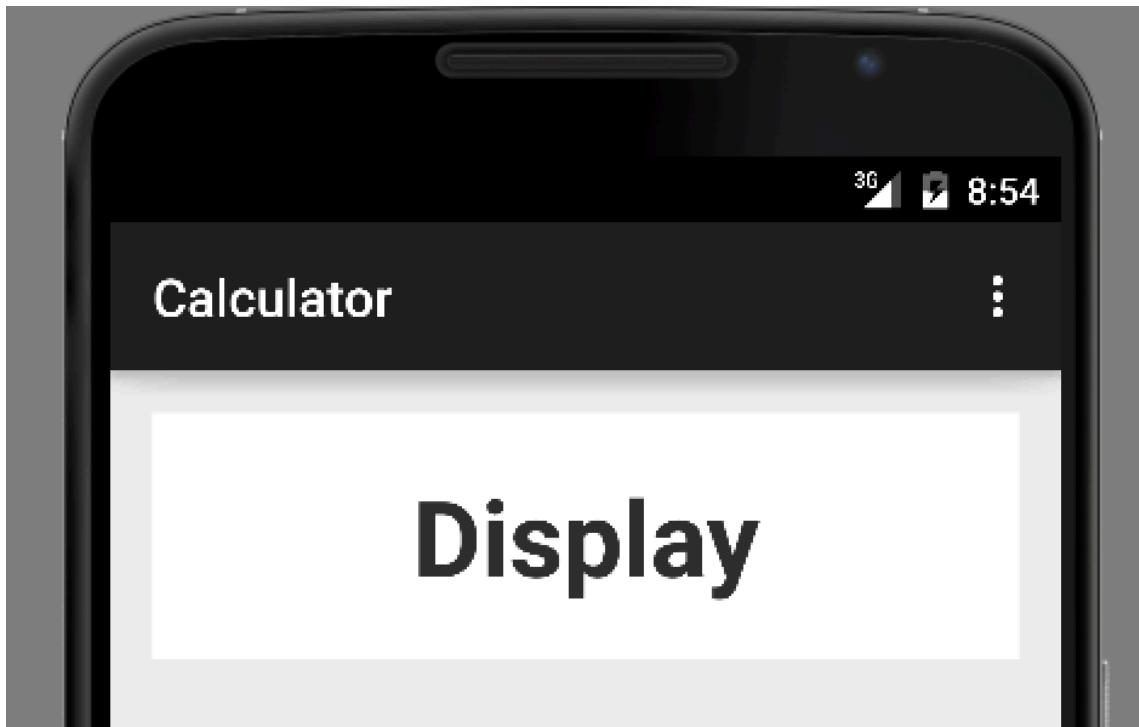
To neaten up the view, remove the `<TextView>` from `activity_calculator.xml` (if present), the string `hello_world` from `strings.xml`, and any tests you may have added for the string in `CalculatorActivityTest`.

Make sure to run all the tests and commit your work.

¹²You may or may not be seeing this text. If you didn't use a wizard to create your project (or used a different setting) it may not have been added to your view. If you added the `<fragment/>` tag after the welcome text in `activity_calculator.xml` it would be hidden behind the display view because there's a `z-order` applied to children of `<RelativeLayout/>`.

Summary

We have a styled display for our results!



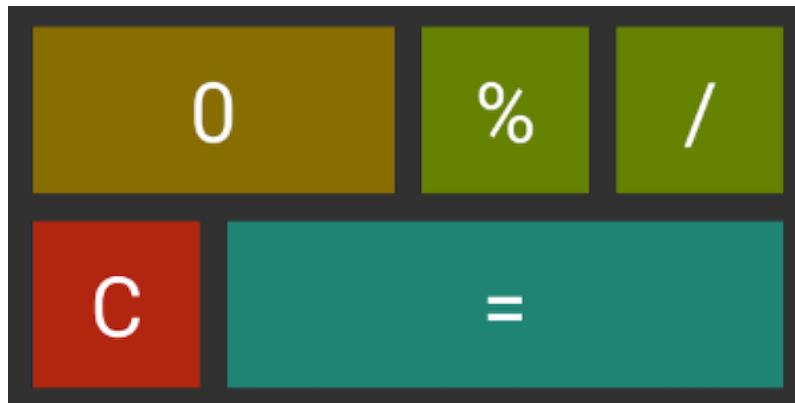
We built a strong foundation that we can iterate on rapidly. As we build out new functionality, we'll add tests and can be sure that we haven't broken anything else along the way.

Now let's add some buttons so we have something to display.

Chapter 5

Building the Buttons

In this chapter, we'll create the buttons for entering numbers and operations. To do this, we create a fragment, `ButtonsFragment` and populate it with many buttons. We host this fragment in the `CalculatorActivity` below the `DisplayFragment`. Finally, after all the views are in place, we'll lay out the buttons.



Before we write code, we write a test. The goal is to write a small test that forces you to expand your application just enough to move forward. Remember, before moving on to new things, tests should pass for the current component. All the tests in the project should pass before you commit.

We follow a similar process to [Chapter 4](#) and reference shortcuts and dialogs as we work through building the buttons.

5.1 Set Up Tests

Create Test File

First, create a file called `ButtonsFragmentTest` in the module `robolectric_tests` by right-clicking on the package name `com.greenlifesoftware.calculator` and select “New” and then “Java Class.” After entering the name, click “OK.”

Generate the first test, `shouldBeNull()`, using **Command N**.

Create a new instance of the `ButtonsFragment` class using the factory pattern (See [Refactor: Factory Pattern](#))¹ and assert that `ButtonsFragment` is non-null.

Use **Option Enter** to import `org.junit.Assert.assertNotNull`, generate the fragment class for you (in the `app` module), and generate the `newInstance()` function in the `ButtonsFragment` class.

After this, you should have a `ButtonsFragment` class (see [Section 5.1](#)) and a test:

```
import static org.junit.Assert.assertNotNull;

public class ButtonsFragmentTest
{
    @Test
    public void shouldBeNull() throws Exception
    {
        assertNotNull( ButtonsFragment.newInstance() );
    }
}
```

¹In the last chapter, we did this in separate steps to introduce you to the logic behind certain testing and architecture conventions.

Extend Fragment

Open the **ButtonsFragment** file, extend **Fragment**, and import the support library.

Now add the empty constructor using the shortcut **Control O**.

Reminder: The [documentation](#) suggests adding this constructor any time you create a **Fragment** subclass.

The class should look like this:

```
import android.app.Fragment;

public class ButtonsFragment extends Fragment
{
    public ButtonsFragment()
    {
    }

    public static ButtonsFragment newInstance()
    {
        return new ButtonsFragment();
    }
}
```

Finish Test Configuration

We need to refactor our test to use Robolectric's `startFragment()` API.

Promote your `newInstance()` call to a field called `fragment`. To do this, use the shortcut **Command Option F** and chose to initialize it in `setUp()`.

In `setUp()`, call `startFragment()` with our `fragment` field.

Finally, add the `@Run` annotation to use our custom test runner.

Your test class should look like this:

```
@RunWith(RobolectricGradleTestRunner.class)

public class ButtonsFrgmentTest
{
    private ButtonsFrgment fragment;

    @Before
    public void setUp() throws Exception
    {
        fragment = ButtonsFrgment.newInstance();
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull(fragment);
    }
}
```

Run the tests for just this class or use the “All Tests” run configuration (see the [Run Configs](#) section). The test passes, commit your work.

5.2 Button Container

Create Test

Now we're ready to start testing the calculator buttons view.

Create a test named `shouldHaveButtons()`. We'll use the `fragment` field to get access to the view and assert that it's visible (and non-`null`).

```
@Test
public void shouldHaveButtons() throws Exception
{
    assertViewIsVisible( fragment.getView()
        .findViewById( R.id.calculator_buttons ) );
}
```

You'll notice a compilation error because we are trying to look up an id, `calculator_buttons`, that doesn't exist yet.

Create View

To construct the buttons view, use the shortcut **Control O** to override the life-cycle method `onCreateView()`. Here we inflate our view using the layout file, `inflater`, and the view `container`. After inflating the view, we return it to the caller.

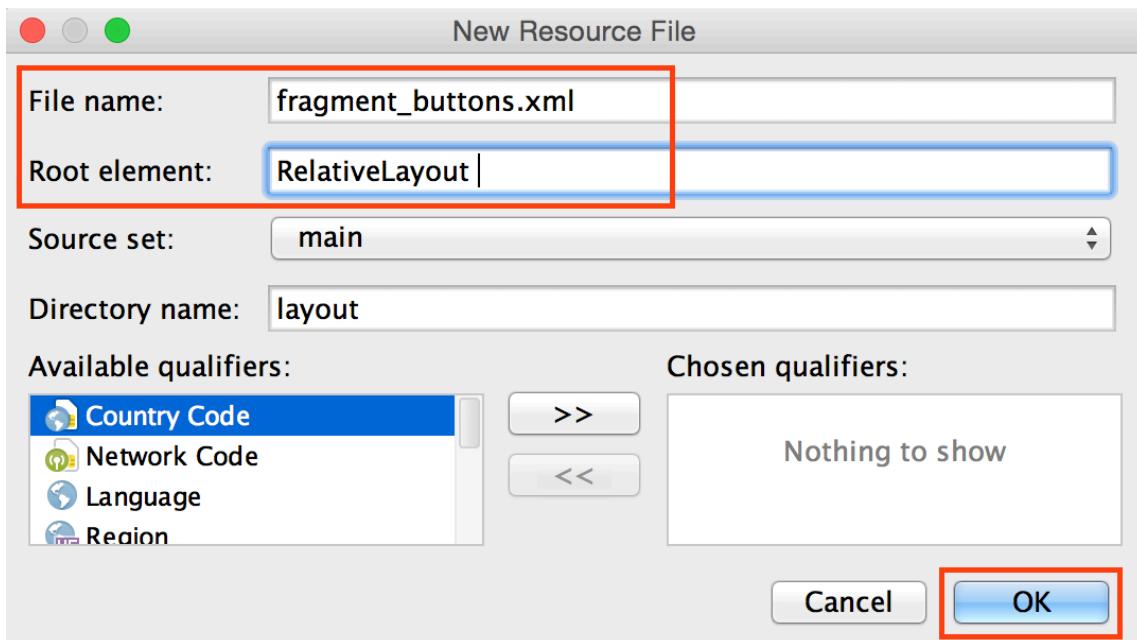
```
@Override
public View onCreateView( LayoutInflater inflater,
    ViewGroup container,
    Bundle savedInstanceState )
{
    View layout = inflater.inflate( R.layout.fragment_buttons,
        container,
        false );
    return layout;
}
```

You'll notice a compilation error because we are trying to use a layout file that doesn't exist.

XML

Use the shortcut **Option Enter** to create the layout file.

When the “New Resource File” dialog pops up, enter **RelativeLayout** as the “Root element” and click “OK.”



Now add the id, **calculator_buttons**, to the **RelativeLayout**. Your XML should look like this:

```
<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/calculator_buttons"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
</RelativeLayout>
```

Run the tests. Commit your work when they pass.

5.3 Add Buttons

In this section, we'll add a button to the view for each number and operator.

The buttons will be ugly and won't do anything. We'll make them pretty in the [Styling the Calculator](#) chapter and make them do something in the [Button Interaction](#) chapter.

Add Button Test

As always, start small. Add a test for just one button.

```
@Test
public void shouldHaveOneButton() throws Exception
{
    assertViewIsVisible( fragment.getView()
        .findViewById( R.id.button1 ) );
}
```

At this stage, the id, `R.id.button1`, doesn't exist, so compilation will fail.

Add Button

Add a view to the `fragment_buttons.xml` layout with an id of `button1`.²³

```
<!--1-->
<Button
    android:id="@+id/button1"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:layout_margin="5dp"/>
```

The tests should pass at this stage.

²The first line is a comment. You can use the key combination **Command /** to create one for you. It uses the correct comment format for your file type.

³We've hard-coded some basic sizing and margins, they will be refactored later.

Verify String

Let's extend our test to ensure that the correct string is configured in the view.

First we'll create a local variable, `button1` and cast to a `Button`. We'll use our `ResourceLocator` class to get the resource string and compare it to the text configured on `button1` to ensure they are equal.⁴

```
@Test  
public void shouldHaveOneButton() throws Exception  
{  
    Button button1 = (Button) fragment.getView().findViewById( R.id.button1 );  
    assertViewIsVisible( button1 );  
    assertThat( button1.getText().toString(),  
                equalTo( getString( R.string.BUTTON_1 ) ) );  
}
```

We can't compile at this point since the string, `BUTTON_1`, doesn't exist.

Add String

It's best practice to move all strings to the `strings.xml` file instead of leaving them hard-coded in the view.⁵

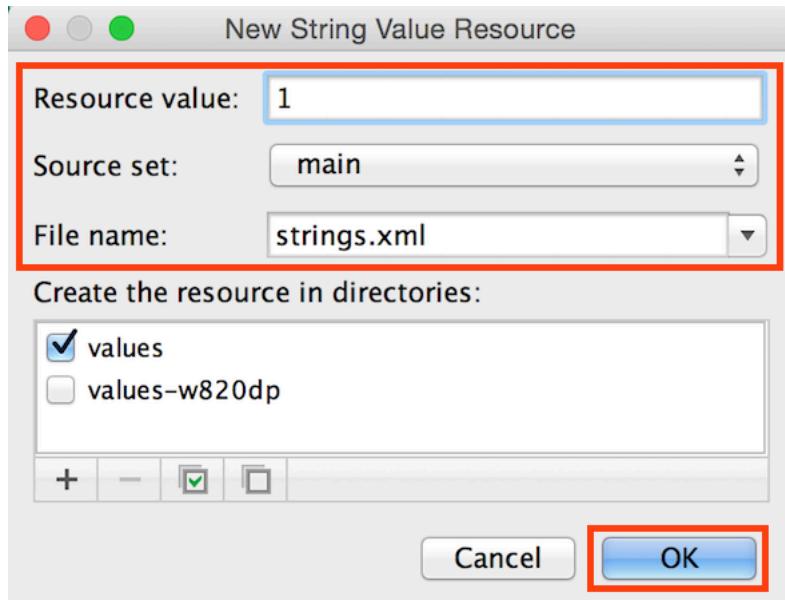
We can generate the string from the layout file by using the shortcut **Option Enter** (when your cursor is on `@string/BUTTON_1`). Click on “Create string value resource ‘BUTTON_1’” or use the keyboard to select it.



⁴Make sure to import the libraries you need.

⁵This simplifies internationalization (Section ??) and string resource testing. Also, it makes you look like you know what you're doing.

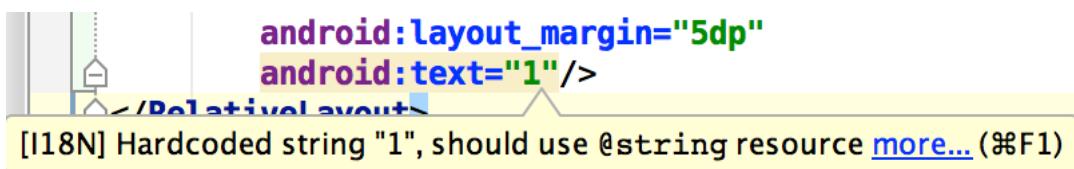
In the dialog, enter “1” as the “Resource value” and click “OK.”



Open the **strings.xml** by using **Command Shift O** and entering **strings.xml** for the file name or by using **Command B** (or **Command Click**) on the name **@string/BUTTON_1**.⁶ You’ll see this line:

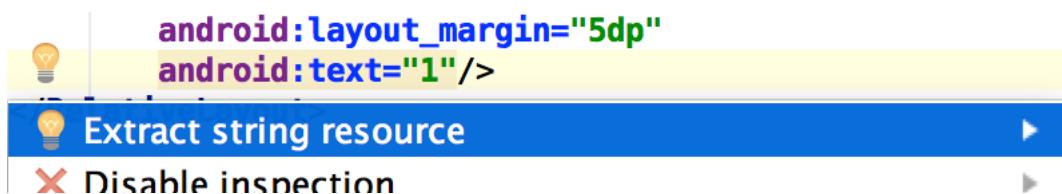
```
<string name="BUTTON_1">1</string>
```

However, if you added the text directly to the view (e.g. instead using a reference to a string resource or typing the reference incorrectly), the lint tool will recommend that you move the string to your **strings.xml** file.



⁶You can navigate back to your last file by using the **Command [** shortcut.

You can use the key combination **Option Enter** to let the IDE abstract this for you. It's basically the same process as before, but this time you specify the name of the id or “Resource name,” instead of the value of the string. It will create a string with whatever name you enter in the dialog.



Add Other Buttons

Follow the process to create each number and operator button. Make sure the tests pass along the way.

- Numbers 0-9
- + (Plus)
- - (Minus)
- * (Multiply)
- / (Divide)
- % (Modulus)
- = (Equals)
- C (Clear)

After you’re done with the buttons, commit. See the section, [Refactor: Get View](#), for a way to simplify your tests.

Refactor: Get View

One thing you'll see frequently is a look up from the fragment's view with an id. Since there's repetition, it's an excellent candidate for refactoring.

Let's create a helper function that takes the view's **id** and fetches a **View** using the **fragment** field.

```
private View getViewById( int id )
{
    return fragment.getView().findViewById( id );
}
```

We can take this one step further since most of the views that we will be fetching are **Buttons**. Create a second helper function that uses **getViewById()** and casts the result to a **Button** before returning.

```
private Button getButtonById( int id )
{
    return (Button) getViewById( id );
}
```

Functions that used **fragment.getView().findViewById()** can be replaced with one of these two calls.

```

@RunWith(RobolectricGradleTestRunner.class)

public class ButtonsFragmentTest
{
    private ButtonsFragment fragment;

    @Before
    public void setUp() throws Exception
    {
        fragment = ButtonsFragment.newInstance();
        startFragment( fragment );
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( fragment );
    }

    @Test
    public void shouldHaveButtons() throws Exception
    {
        assertViewIsVisible( getViewById( R.id.calculator_buttons ) );
    }

    @Test
    public void shouldHaveOneButton() throws Exception
    {
        Button button1 = getButtonById( R.id.button1 );
        assertViewIsVisible( button1 );
        assertThat( button1.getText().toString(),
                    equalTo( getString( R.string.BUTTON_1 ) ) );
    }

    @Test
    public void shouldHaveTwoButton() throws Exception
    {
        Button button2 = getButtonById( R.id.button2 );
        assertViewIsVisible( button2 );
        assertThat( button2.getText().toString(),
                    equalTo( getString( R.string.BUTTON_2 ) ) );
    }

    // ...

    private Button getButtonById( int id )
    {
        return (Button) getViewById( id );
    }

    private View getViewById( int id )
    {
        return fragment.getView().findViewById( id );
    }
}

```

5.4 Show Buttons

So far we've been creating our `ButtonsFragment` in isolation. Let's add the fragment to `CalculatorActivity` so we can see it.

Test Presence

First, add a test named `shouldHaveButtonsFragment()` to `CalculatorActivityTest`.

```
@Test  
public void shouldHaveButtonsFragment() throws Exception  
{  
    assertNotNull( activity.getSupportFragmentManager()  
                    .findFragmentById( R.id.buttons_fragment ) );  
}
```

It doesn't compile, without the fragment in `activity_calculator.xml`.

Add Fragment

Add our fragment `ButtonsFragment` below the `DisplayFragment`. We use `RelativeLayout`'s ability to position child views with the attribute `layout_below`.

```
<!--Buttons-->  
<fragment  
    android:id="@+id/buttons_fragment"  
    android:name="com.greenlifesoftware.calculator.ButtonsFragment"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:layout_below="@+id/display_fragment" />
```

The tests pass after adding the fragment.⁷

⁷There are two fragment managers for this activity. Support fragments live in the support fragment manager (retrieved with `getSupportFragmentManager()`) and framework fragments live in

Refactor

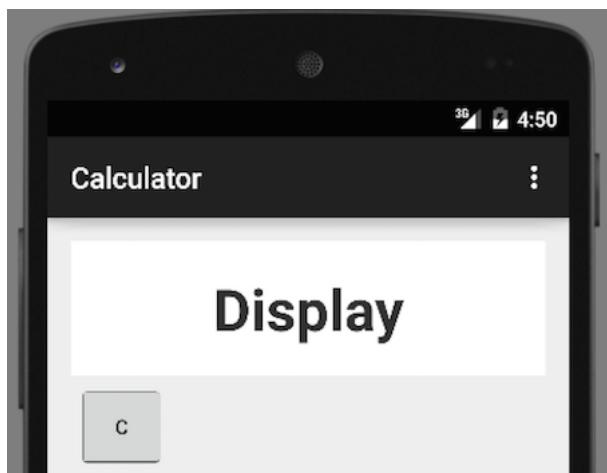
We have two tests that are fetching the fragment manager, so let's make a helper function, `getFragmentById()` that returns a `Fragment` based on an id. Update both tests to use the helper function.

```
@Test
public void shouldHaveButtonsFragment() throws Exception
{
    assertNotNull( getFragmentById( R.id.buttons_fragment ) );
}

Fragment getFragmentById( int id )
{
    return activity.getSupportFragmentManager().findFragmentById( id );
}
```

View App

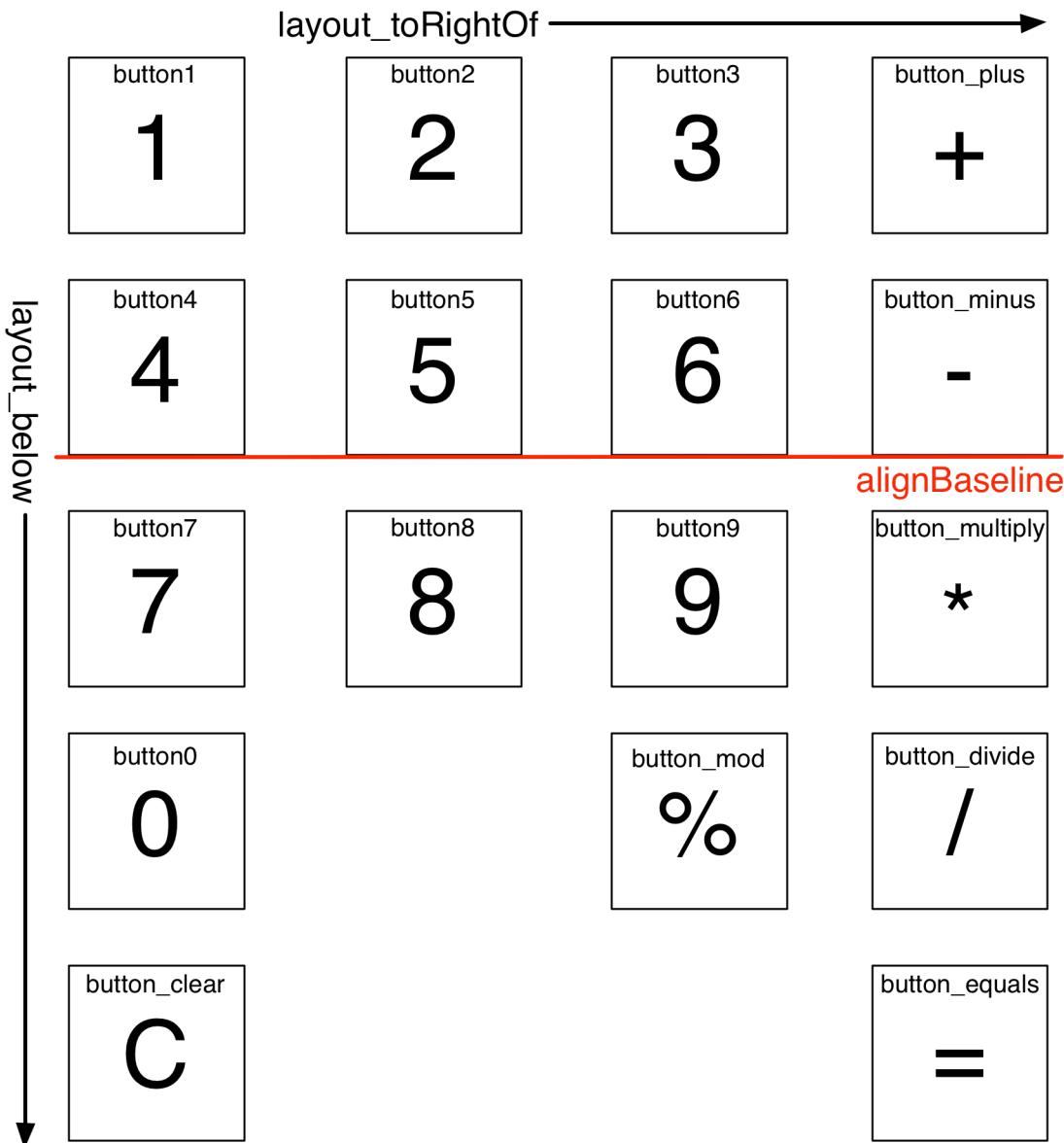
When we run the app, the view doesn't look great. All the buttons are stacked on top of each other (but at least they are under the display). We'll make them all visible in the next section.



the regular fragment manager (retrieved with `getFragmentManager()`). If your test fails, you may have imported the wrong `Fragment` type in your `ButtonsFragment`.

5.5 Layout the Buttons

We'll use `RelativeLayout`'s attributes to define the position of the buttons relative to other buttons in the view.⁸



⁸There are many ways this view could be laid out. `RelativeLayout` isn't the most intuitive choice, but it is quite useful and mastery of it early in your Android career will lead to great things.

I've added the first row and beginning of the second here for reference.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/calculator_buttons"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!--1-->
    <Button
        android:id="@+id/button1"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_margin="5dp"
        android:text="@string/BUTTON_1"/>

    <!--2-->
    <Button
        android:id="@+id/button2"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_margin="5dp"
        android:layout_toRightOf="@+id/button1"
        android:text="@string/BUTTON_2"/>

    <!--3-->
    <Button
        android:id="@+id/button3"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_margin="5dp"
        android:layout_toRightOf="@+id/button2"
        android:text="@string/BUTTON_3"/>

    <!--4-->
    <Button
        android:id="@+id/button4"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_margin="5dp"
        android:layout_below="@+id/button1"
        android:text="@string/BUTTON_4"/>

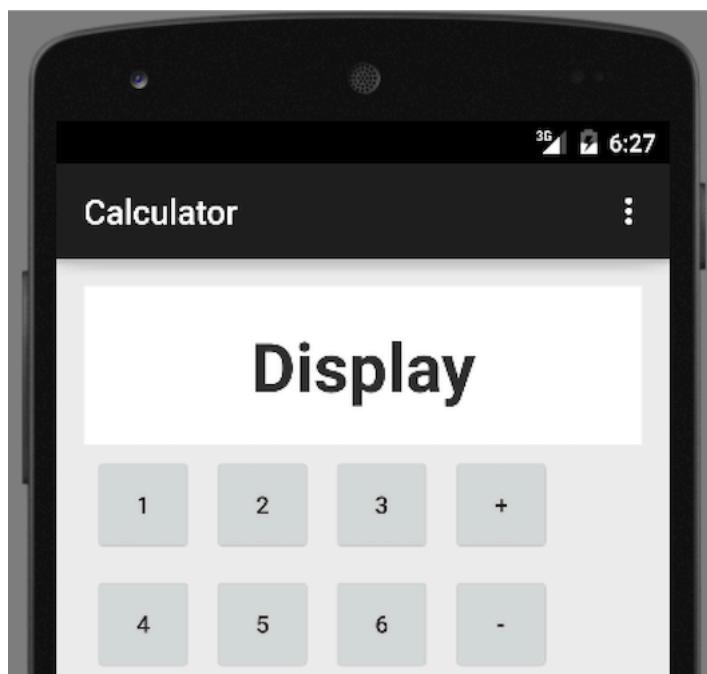
    <!--5-->
    <Button
        android:id="@+id/button5"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_margin="5dp"
        android:layout_toRightOf="@+id/button4"
        android:layout_alignBaseline="@+id/button4"
        android:text="@string/BUTTON_5"/>
    ...
</RelativeLayout>
```

Note: `RelativeLayout` doesn't allow forward declarations of ids. This means that an id you plan to use in an attribute has to be defined before being used. Depending how you specify your view, you may need to move some view elements around to accommodate this.

After you've made all the buttons visible, run the tests. If they pass, commit your work.

Summary

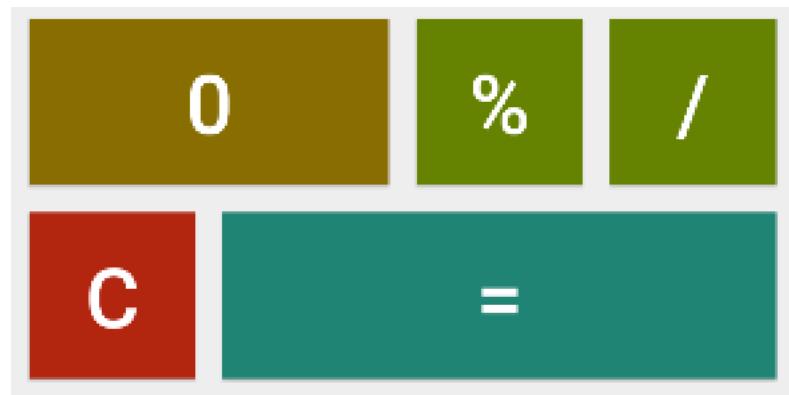
Now we have buttons *and* a display. Next we'll learn about the resource system and how to use it to improve the visual representation of our buttons and our code base at the same time.



Chapter 6

Styling the Calculator

In this chapter we'll add some style to the calculator. First we'll apply styles to all of our buttons and the display. Next we provide interaction when users click on the newly styled buttons. Finally, we'll add an icon for the app.



You're welcome to update the colors to meet your needs. Colors have been provided for your use in RGB. If you'd like to define different colors, use the color picker tool or different RGB values.

Note: Although we won't be adding tests for the activities in this chapter, you should still run tests when you make changes. This helps you catch errors earlier in the process. I always run the tests for the whole project just before a commit.

6.1 Button Styles

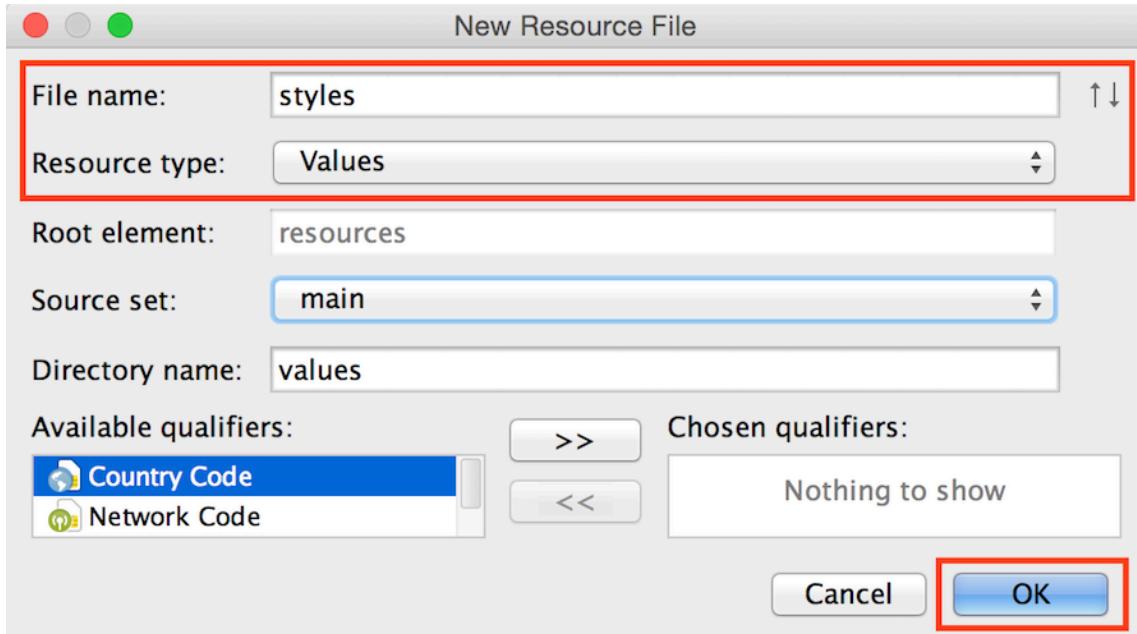
If you analyze the design, you'll notice that the buttons are variations on a theme. The only differences are colors and some buttons are wider than others.

Since these buttons are so similar, it makes sense to create a basic style that each button type will inherit.

We'll use a style file to manage our styles and themes.

To create this file from scratch,¹ right-click on the `res/` directory in your `app` module. Select “New” and “Android resource file.”

Enter “styles” in the “File name” box and ensure that “Resource type” is set to “Values”. Click “OK.”²



This creates a `styles.xml` file in `app/src/main/res/values/`. We'll add a style in the next section.

¹If you used the project wizard, the file already exists and has one style, `AppTheme`.

²If you try to create the file when it already exists, you get this error message

```
Cannot create 'app/src/main/res/values/styles.xml'. File already exists.
```

Base Button

Since the buttons are similar, we will create a base style and later tweak for different button types.

Create a style named **BaseButton**:

```
<resources>
    ...
    <style name="BaseButton">
        <item name="android:layout_width">@dimen/button_dimensions</item>
        <item name="android:layout_height">@dimen/button_dimensions</item>
        <item name="android:layout_margin">@dimen/button_margin</item>
        <item name="android:textSize">@dimen/button_text_size</item>
        <item name="android:textColor">@android:color/white</item>
    </style>
</resources>
```

Specify the height and width of the button, the margin between them, the color of the text, the size of the font, and the text color. For **textColor** we use an Android supplied color.

Note: The attributes here are in a slightly different format than you'd find in an layout file, but they have the same affect.

Most attributes use abstracted **dimens**, which we'll add in the next section.

Create Dimensions

Let's create the dimensions we referenced in section [Base Button](#). Create and/or open a file called `dimens.xml` in `app/main/res/values/`.

If this file doesn't exist, use the same process we followed in section [Button Styles](#) to create the file (use "dimens" as the file name instead of "styles"). If you used the project wizard, this file already exists and will have a sister file in `app/src/main/res/values-w820dp`.

Enter these button dimensions:

```
<resources>
    ...
    <!--Buttons-->
    <dimen name="button_margin">5dp</dimen>
    <dimen name="button_dimensions">60dp</dimen>
    <dimen name="button_text_size">30sp</dimen>
</resources>
```

Note: We use `dps` for view sizing and `sps` for text sizing. You can use `dps` for the text, but that prevents your users from adjusting their font sizes. Using `sp` makes your application accessible to more users.

The values added to `values/dimens.xml` are the general, or fall-back, case. When Android can't find a resource qualifier more specific to use (e.g. `values-w820dp`), it will use the value here. If you don't specify a value in this file, your app will crash at runtime when it tries to reference the missing value.³

We'll add dimensions for other screen sizes in the section [Flexible Dimensions](#).

³It's possible to define a dimension in `values-w820dp` and not realize you didn't put one in `values`. The IDE won't complain because it's been specified somewhere. You might not notice if your testing device is wider than `820dps`.

Number Style

Now that we have **BaseButton** and the dimensions we need, let's use them.

Add a new style to **styles.xml** called **Number**.

Add the **parent** attribute to inherit from the **BaseButton** style. This means we get all the attributes we defined in the **Base Button** section. We can extend the style by overriding inherited attributes or adding new ones.

```
<!--Number-->
<style name="Number" parent="BaseButton">
    <item name="android:background">@color/brown</item>
</style>
```

In our **Number** style, we leave the the inherited attributes alone and add one for the background color.

Add the color **brown** to the **colors.xml** file using the RGB value **#876D13**.

```
<color name="brown">#876D13</color>
```

Later, in the **Button State** section, we use this background attribute to add visual feedback when a user touches a button.

Now that we have the style, let's apply it to the number buttons.

Apply Style

To use the **Number** style, open the layout file `fragment_buttons.xml` and add the **style** attribute to a view.

```
<!--1-->
<Button
    android:id="@+id/button1"
    android:text="@string/BUTTON_1"
    style="@style/Number"/>
```

Note: The **style** attribute doesn't use the **android** namespace like other attributes in our view.

Now remove attributes that are duplicated by the **Number** style: `layout_width`, `layout_height`, and `layout_margin`.

```
<!--6-->
<Button
    android:id="@+id/button6"
    android:layout_toRightOf="@+id/button5"
    android:layout_alignBaseline="@+id/button5"
    android:text="@string/BUTTON_6"
    style="@style/Number"/>
```

It's best practice to move as much as possible to the styles to keep your XML files lean. You should leave attributes that are specific to the button, such as view positioning (`layout_toRightOf` and `layout_alignBaseline`), the view's `id`, and the button's `text`.⁴

Styles allow us quickly change the view. For example, let's say you wanted to make the buttons twice as big. When we didn't have a style, the sizing was hard-coded on every button. You'd need to change `layout_width` and `layout_height` of all nine buttons. However, since we moved this to a dimension (that we use in a style), you only need to change the value of `button_dimensions` in `dimens.xml`.

⁴Check out these [interesting thoughts](#) on the topic.

Other Buttons

Now let's create styles for the rest of the button types and apply them in the layout file `fragment_buttons.xml`. When you finish, the view should look like the screenshot in the introduction of this chapter.

Zero

This button is similar to the other number keys, but it's twice as wide.

1. Create style `Zero` that inherits from `Number`.
2. Override the button width by adding the `layout_width` attribute.
3. Add a dimension `zero_button_width` with the value `130dp` to `dimens.xml`.

```
<!--Zero-->
<style name="Zero" parent="Number">
    <item name="android:layout_width">@dimen/zero_button_width</item>
</style>
```

Operator

1. Create style `Operator` that inherits from `BaseButton`.
2. Add a background color with the `background` attribute.
3. Add the color `green` with the the RGB value `#668113` to `colors.xml`.

```
<!--Operator-->
<style name="Operator" parent="BaseButton">
    <item name="android:background">@color/green</item>
</style>
```

Clear

1. Create style **Clear** that inherits from **BaseButton**.
2. Add a background color with the **background** attribute.
3. Add the color **red** with the RGB value **#B0281A** to **colors.xml**.

```
<!--Clear-->
<style name="Clear" parent="BaseButton">
    <item name="android:background">@color/red</item>
</style>
```

Equals

1. Create style **Equals** that inherits from **BaseButton**.
2. Override the button width by adding the **layout_width** attribute.
3. Add a dimension **equals_button_width** with the value **200dp** to **dimens.xml**.
4. Add a background color with the **background** attribute.
5. Add the color **blue** with the RGB value **#268474** to **colors.xml**.

```
<!--Equals-->
<style name="Equals" parent="BaseButton">
    <item name="android:layout_width">@dimen>equals_button_width</item>
    <item name="android:background">@color/blue</item>
</style>
```

6.2 Display Style

Finally, let's create a style and clean up the layout file, `fragment_display.xml`.

```
<EditText
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/calculator_display"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/DEFAULT_DISPLAY"
    style="@style/Display">
</EditText>
```

Create a style named `Display` and move most of the attributes here.

```
<!--Display-->
<style name="Display">
    <item name="android:paddingTop">@dimen/display_padding</item>
    <item name="android:paddingBottom">@dimen/display_padding</item>
    <item name="android:textSize">@dimen/display_text_size</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">@color/nearly_black</item>
    <item name="android:maxLength">@integer/display_max_length</item>
    <item name="android:editable">false</item>
    <item name="android:inputType">none</item>
    <item name="android:focusable">false</item>
    <item name="android:focusableInTouchMode">false</item>
    <item name="android:background">@android:color/white</item>
</style>
```

Note: You can abstract more than just dimensions and colors. Here we abstract an integer.

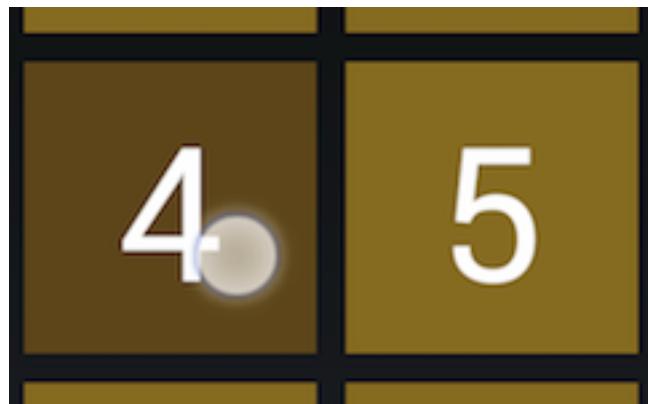
6.3 Button State

It's best practice to provide visual feedback to users when they interact with your interface.

When you use default Android components (e.g. [Buttons](#)), a default state list drawable is associated with the view. This drawable specifies how to react to different button interactions.

We lost this builtin visual feedback when we changed the background to a flat color. We can add it back by defining our own states.

We set a different color for when a button is pressed, when it is focused,⁵ and its default state.



Since we are using flat colors (defined in the next section) as our button “images,” there is no need to provide [png](#) or [jpg](#) images in different densities. A color scales well enough on its own. If you use images in future applications, whenever possible, use 9-patches.

⁵It improves accessibility to handle the focused state (e.g. using a scroll wheel or arrow keys on a physical keyboard) of buttons in your application. Although accessibility is out of the scope of this book, you should take it into consideration when creating your applications. Android has [suggestions](#) and a [checklist](#) to help you with this effort.

Colors

To add visual feedback, we use color variations for the different button actions.

In `colors.xml`, create one color that represents a slightly darker selected state for each of the button types: number, operator, equals, and clear.

Add a yellow color for the button's focused state.

```
<!--General-->
<color name="focused_yellow">#AFB315</color>

<!--Clear-->
<color name="red">#B0281A</color>
<color name="selected_red">#7D1918</color>

<!--Operators-->
<color name="green">#668113</color>
<color name="selected_green">#3F4F0d</color>

<!--Equals-->
<color name="blue">#268474</color>
<color name="selected_blue">#144A40</color>

<!--Numbers-->
<color name="brown">#876D13</color>
<color name="selected_brown">#604613</color>
```

The default, or normal state, was already specified for each button.

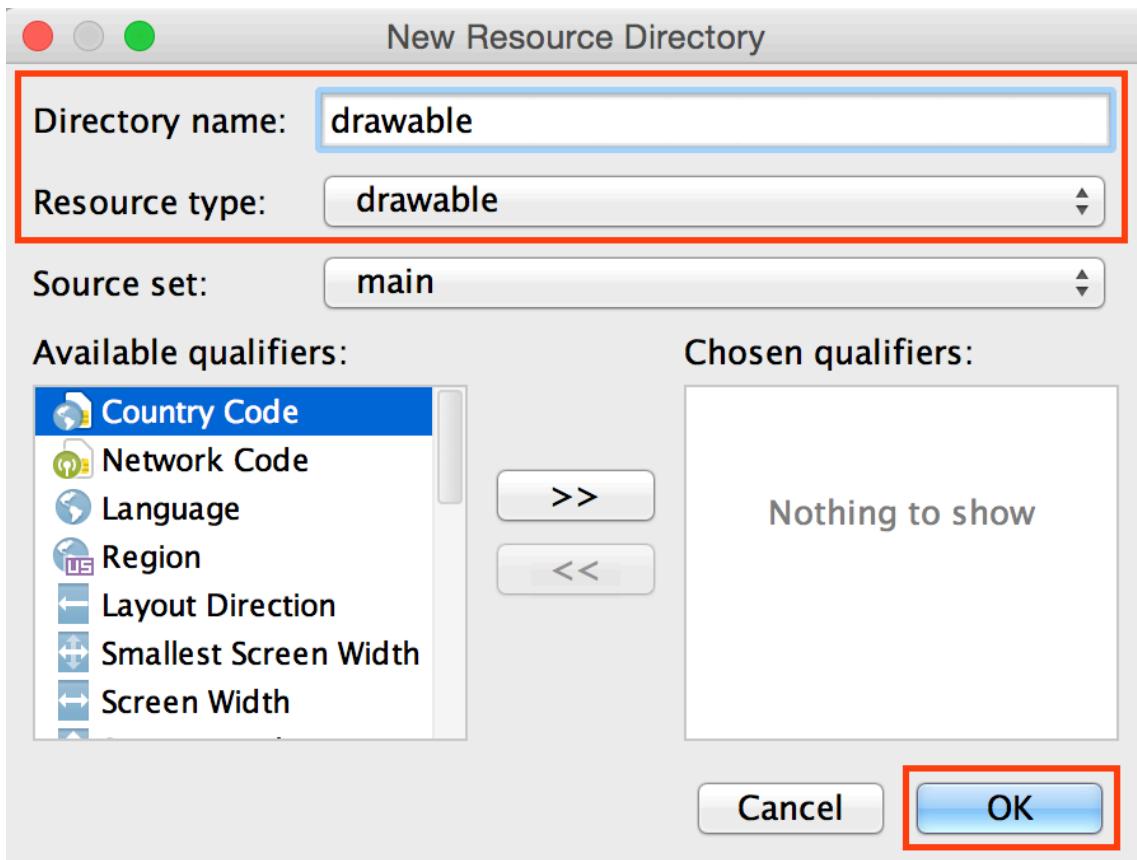
Note: This color is currently set as the button's background color in `styles.xml` and we will update it soon.

Drawable Folder

Now that we have colors, we will create the state list drawables.

First, create a non-qualified directory called `drawable/` under `res/`. We'll add our selector files for each button type to this directory.

To create it, right-click on the `res` directory and select “New” -> “Android resource directory.”

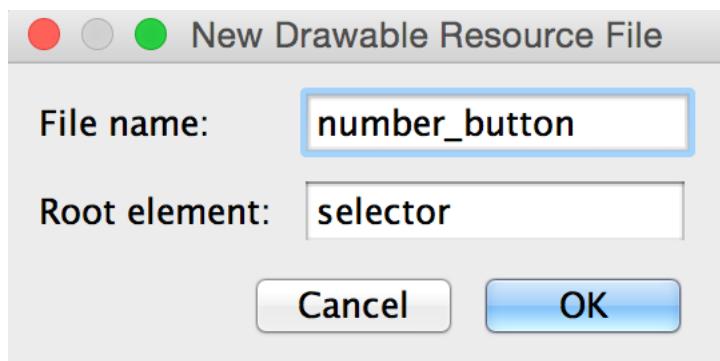


Specify the “Resource type” and “Directory name” as `drawable` and click “OK.” This directory represents the fall back case for fetching drawables (similar to the function of the `values` directory for `dimens.xml`).

Selector

To create the selector for our number buttons, create a file named `number_button.xml` by right-clicking on the `drawable` directory and selecting “New” -> “Drawable resource file.”

In the dialog, enter `number_button` as the “File name” and `selector` as the “Root element.” Click “OK.”



Add three states, `state_pressed`, `state_focused`, and an unnamed state. The last state should be at the bottom of the file.

Associate the darker colors we defined with the selected state, the yellow with focused, and the base color with the normal state (in the `drawable` attribute).

```
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">

    <!--Pressed-->
    <item
        android:state_pressed="true"
        android:drawable="@color/selected_brown"/>

    <!--Focused-->
    <item
        android:state_focused="true"
        android:drawable="@color/focused_yellow"/>

    <!--Normal-->
    <item
        android:drawable="@color/brown"/>
</selector>
```

Apply Selector

Now that we have a selector, we need to associate it with the number buttons.

Open `styles.xml` and find the `Number` style. The `background` attribute is currently set to a specific color.

Update the `background` attribute to use `number_button.xml`, our new selector file. Since it's a drawable (e.g. lives in one of the the `drawable` directories), we can use it like any other drawable.

```
<!--Number-->
<style name="Number" parent="BaseButton">
    <item name="android:background">@drawable/number_button</item>
</style>
```

Now when a user clicks on a number button, they get a visual response.

Other Buttons

Now that we have the number buttons responding to feedback, create selectors for the other buttons: equals, operator, and clear.

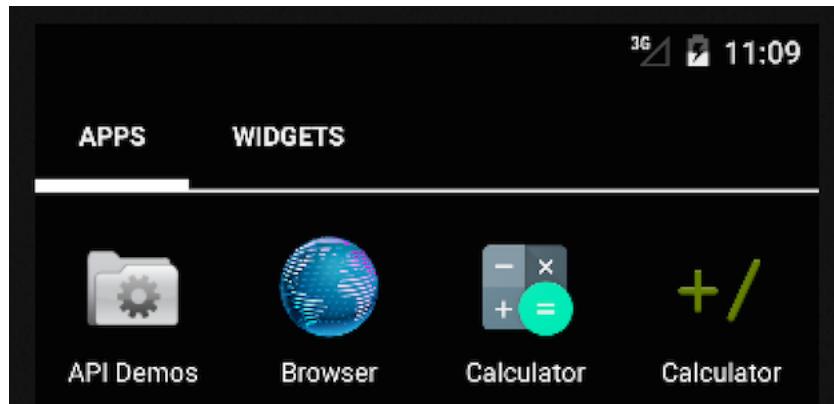
Create a selector with the new colors we specified. Then update the associated style to use the new selector.

When you are done, build and run your application on a device. Ensure that each button color gets darker when pressed or yellow when selected.

When you are happy with the results, run the tests. When the tests for the project pass, commit your work.

6.4 App Icon

In this section, we will create an icon to be shown in the launch grid on your device. The same icon will be shown if you create a shortcut to the application.



Create Icon

You can create the icon with your favorite application⁶, but we'll use [Android Asset Studio](#). Choose “Launcher icons” to start designing your icon. Make something fun and download the zip file (bottom of the screen) when you’re done.

A screenshot of the "Android Asset Studio" website, specifically the "Launcher Icon Generator" section. At the top left, there is a back arrow labeled "< ANDROID ASSET STUDIO". The main title is "Launcher Icon Generator". Below the title, there is a "Foreground" input field containing the text "Foreground". To the right of the input field is a button bar with three options: "IMAGE", "CLIPART", and "TEXT", with "TEXT" being the selected option (indicated by a blue background). Below the input field, there are two rows of controls. The first row has a "TEXT" label and a text input field containing the text "+/". The second row has a "FONT" label and a text input field containing the font name "Courier New". To the right of these controls is a preview area showing a white square containing a black plus sign and a black diagonal slash. The entire interface is set against a light gray background.

⁶It can get complicated to create images in the right dimensions. Start with this tool and create fancy icons later. For more information, watch [this video](#) and then check out [this overview](#) and [follow up article](#).

Integrate Images

When you unzip the file, it will create a folder named `res/`. Inside there's several images in qualified folders (e.g. `drawable-mdpi`).

If you used the project wizard to create your project files, you likely already have the folders in your `app/src/main/res/` directory.

When the directory already exists, move the image to its correct folder (e.g. move the downloaded `res/drawable-mdpi/ic_launcher.png` to your project directory `app/src/main/res/drawable-mdpi/`). When prompted, override any existing icons.

If the directory doesn't already exist (e.g. `drawable-xxxhdpi`), move the entire folder to `app/src/main/res/` instead of just the `.png` file.

Note: The tool doesn't provide a resource for `drawable-ldpi`. Currently, only 5.9% of devices are using this resolution.⁷

Android Manifest

To add the icon to your app, specify it in `AndroidManifest.xml` in the `app` module. The drawable resource is added to the `android:icon` attribute in the `<application/>` tag.

```
<application
    android:label="@string/app_name"
    android:icon="@drawable/ic_launcher">
```

Note: If you used the wizard, it's already specified correctly.

Run your application to see the newly configured icon. Commit your work.

⁷The [Android Dashboard](#) is a great place to check when deciding what to support.

Summary

Users respond to good design. Creating a relatable design draws users in and keeps your users engaged.

Start with platform standards and default widgets until they no longer support your vision. Make sure to carefully consider any deviations so that you don't confuse your users or harm the user's ability to use the app.

In this chapter, we learned how to style the display. The styles and dimensions we created make it easy to extend the design to other form factors. We expand on our work in this chapter when in the [Scaling the View](#) chapter. Had we hardcoded these attributes instead, significant refactoring would have been required to support new screen sizes.

We also added visual feedback when users interact with the calculator buttons. Users would feel confused and may not trust that your application is working properly if there was no feedback when touching the buttons (aka [percieved performance](#)). It is important to ground your user in your application, respond to user feedback as quickly as possible, and make the user interface feel tangible.

Finally, we created an icon for our application. Don't underestimate the power of icons – they are an important piece of the user's experience. They help your users recognize your app in a field of icons and convey the essence of your app.

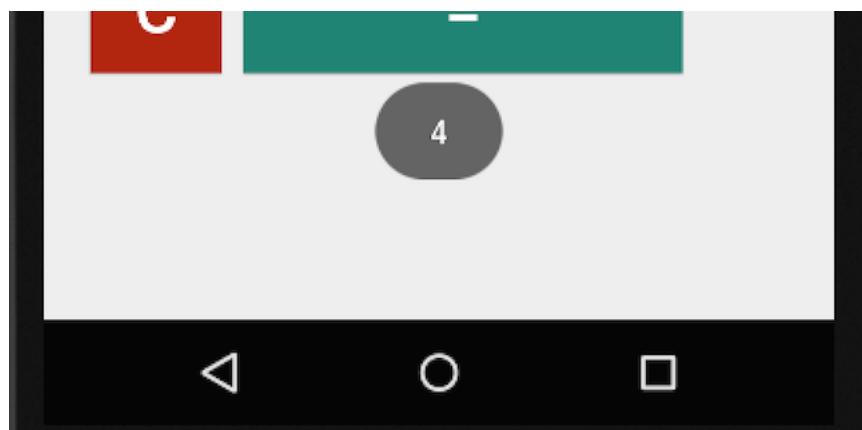
Chapter 7

Button Interaction

Currently our users have buttons that respond to clicks, but these clicks don't affect other aspects of the application. Let's give them some satisfaction.

The first step to adding interactivity to a view is adding a click listener. Any Android component that inherits from `View` (e.g. `TextView` or `Button`), can be turned into a "button" by [adding a click listener](#).

We'll start by showing a `Toast`, which is a brief message that flashes at the bottom of the screen when a user clicks on a number button. To keep it simple, we will show the text configured on that button (e.g. clicking the 1 button will show "1" at the bottom of the screen).



7.1 Toast Test

Let's start by writing a failing test called `oneButtonShouldToast()`. This test compares the button's text to the text of the `Toast` after a click.

We will use the button and its associated text in the comparison. Since we already tested some aspects of this button in the test `shouldHaveOneButton()`, we can use that variable again.

To do this, promote the local variable, `button1`, to a field and initialize the new field in `setUp()`.¹ Now we can reuse this button in any test that needs it instead of redefining it multiple times.

```
private Button button1;

@Before
public void setUp() throws Exception
{
    ...
    button1 = getButtonById( R.id.button1 );
}
```

Note: The test for `shouldHaveOneButton()` should not be failing after the field promotion. If it is, ensure that the field is initialized in `setUp()` and that your test uses the new field correctly.

The next bit of information we'll need is the text that was shown in the `Toast`. Android doesn't provide a way to retrieve this information, but Robolectric's `ShadowToast.getTextOfLatestToast()`² does.

¹The fastest way to create the field is to use the shortcut **Command Control F** and select the place it should be initialized in the pop-up.

²You'll need to import the `library` into your class.

Now that we have access to the information that we're comparing, we will use Robolectric's `performClick()` to simulate a click on the button. This will invoke a `Toast` that will be recorded by Robolectric's internal system and accessible through the call to `getTextOfLatestToast()`.

```
@Test  
public void oneButtonShouldToast()  
{  
    button1.performClick();  
    assertThat( ShadowToast.getTextOfLatestToast() ,  
                equalTo( button1.getText() ) );  
}
```

After the click is performed, we check whether the toasted text and the text of the button are equal.

Run all the tests in the `ButtonsFragmentTest` class. Your new test should be the only one failing because we haven't added a click listener to the button.

```
java.lang.AssertionError: Expected: "1" but: was null
```

We'll do that next.

7.2 Add Click Listener

To add a click listener, open the `ButtonsFragment` class. With `Fragments`, we configure view elements in the `onCreateView()` lifecycle method.

We use the `layout` variable,³ `findViewById()`, and the button's id, `R.id.button1`, to get a reference to the button. Then we add a click listener⁴ using `setOnClickListener()` and create a new `View.OnClickListener()`.

```
layout.findViewById( R.id.button1 )
    .setOnClickListener( new View.OnClickListener()
{
    @Override
    public void onClick( View v )
    {
        Toast.makeText( getActivity(),
            ( (Button) v ).getText(),
            Toast.LENGTH_SHORT )
            .show();
    }
} );
```

When you create a click listener, you need to override the `onClick()` method.⁵ Inside the `onClick()` method, we create a `Toast` using `makeText()` and show it on the screen using `show()`. We pass `makeText()` a context, a string to show,⁶ and the length of time to display the toast.

The call to `makeText()` requires a context. A fragment is not a `Context`, but an `Activity` is. Since every `Fragment` must be hosted by an `Activity`, you can use the `Fragment`'s reference to its hosting parent with `getActivity()`.

Now that the button does something, the tests should pass. Run them and when they pass, commit your work.

³The first line of `onCreateView()` inflates the layout and gives us access to the inflated views.

⁴This is called an anonymous inner class because we're creating a class inline without naming it.

⁵Use the automatic completion tools in the IDE to remove some of this boilerplate.

⁶We are fetching the text from the view by casting the `View`, `v`, to a `Button` and using `Button`'s function `getText()`.

7.3 Refactor Tests

We could keep adding tests manually, but that would get quite repetitive.

In each test, we click the button and compare the text of the toast with the button text. Since this would be repeated almost verbatim for each test, it makes sense to move this to a verification function.

Note: There are a many ways you could compose or abstract a function, class, etc. Some designs are better than others for maximizing maintainability, readability, or loose coupling. When two approaches produce the same result, they are called *functionally equivalent*.

Let's create a function called `verifyToastAfterButtonClick()`.⁷ The only piece of information it will need to complete its work is the button, so let's pass that as a parameter.

```
void verifyToastAfterButtonClick( Button button )
{
    button.performClick();
    assertThat( ShadowToast.getTextOfLatestToast(),
                equalTo( button.getText() ) );
}
```

Move all the code from your test here and use `button` instead. Update the `oneButtonShouldToast()` test to call this new function with `button1`.

```
@Test
public void oneButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button1 );
}
```

⁷This name is a bit long, but you want to convey that the function is doing something and asserting something afterwards. This is an example of increasing readability with smart function naming.

Test Process

For each number button (building on [Toast Test](#)):

1. Promote the button variable to a field
2. Add a test for the button
3. Call the new validation function with the field
4. Add code to the fragment to make the test pass

You could reduce duplication by creating an array to hold our buttons and call `verifyToastAfterButtonClick()` on each one. Since we'll be doing more things to the buttons later, I like to leave things in a fairly tidy state – but not so tidy that I'll have to go unravel things later.

With TDD, you create one test and add the code to make it pass in the fragment before moving on to the next button. For more information about how to make this process smoother, read the next section.

7.4 Refactor Fragment

To keep our `onCreateView()` tidy, create a function to initialize all the number buttons. After creating the function, `configureNumberButtons()`, call it from `onCreateView()`, just after the view inflation.

Promote `layout` to a class field so that we can access it all of our configuration functions.

Note: Make sure to set `layout` to `null` in the `onDestroy()` lifecycle method to avoid memory leaks.

```
private View layout;

@Override
public View onCreateView( ... )
{
    layout = inflater.inflate( ... );
    configureNumberButtons();
    return layout;
}
```

Since the work to configure each button is nearly identical for each button, it makes sense to create another function, `configureNumberButtonWithId()`.⁸ We pass the id we're configuring to this function.

```
private void configureNumberButtons()
{
    configureNumberButtonWithId( R.id.button1 );
}
```

⁸Reminder: use code completion and **Option Enter** to generate code for you as you type.

For each button, we get a reference to it (using `layout` and the id of the button) and add a click listener. The click listener is identical for each button (since it simply shows the text on the button after a click), so create another function called `createNumberOnClickListener()` and set it by using the call to `setOnClickListener()`.

```
private void configureNumberButtonWithId( int id )
{
    layout.findViewById( id )
        .setOnClickListener( createNumberOnClickListener() );
}
```

In this final function, we create and return a click listener (which does the same work that we defined in the [Add Click Listener](#) section).

```
private View.OnClickListener createNumberOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity(),
                ((Button) v).getText(),
                Toast.LENGTH_SHORT )
                .show();
        }
    };
}
```

Delete the code related to the one button configuration from `onCreateView()` and use this series of function calls instead.

Before moving on to new buttons, run the tests to make sure everything is working after your refactor. Commit your work.

7.5 Finish Number Buttons

Now you’re ready to ping pong between the tests and the fragment to configure each button. Follow this process to build out your tests and buttons:

1. Write a failing test as described in the [Test Process](#) section.
2. Add a call to `configureNumberButtonWithId` with the button’s id in `configureNumberButtons ()` function to make it pass.
3. Repeat steps 1 & 2 until each button is configured.
4. After all buttons are configured, run all the tests in the project. If there are failing tests or build errors, fix them.
5. Run the application.
6. Click on each number button to see a toast of the button’s text.

After you’ve completed all the number buttons, the tests pass, and the button clicks show a toast, commit your work.

7.6 Configure Operator Buttons

We will follow a similar process to test and configure the operator buttons.

First, add an operator verification function (like we did in Refactor Tests).

```
void verifyToastAfterOperatorClick( Button button )
{
    button.performClick();
    assertThat( ShadowToast.getTextOfLatestToast() ,
                equalTo( button.getText() ) );
}
```

Note: Although the code is identical to number button verification, we will be treating operators and numbers differently in the next few chapters. We separate them here to make that transition easier.

Promote each operator to a field (**plus**, **minus**, **multiply**, **divide**, **modulo**) and initialize them in **setUp()** so that you can use them throughout the tests.

Now add a test for each operator and use the verification helper. An example for the plus button is shown below.

```
@Test
public void plusButtonShouldToast() throws Exception
{
    verifyToastAfterOperatorClick( plus );
}
```

Configure Buttons Fragment

To make the tests pass, we configure a click listener in `ButtonsFragment`. The set up is similar to the number buttons we configured in the [Add Click Listener](#) section.

Create a function called `configureOperators()` and call it from `onCreate()`. Create a helper function called `configureOperatorWithId()` that takes the id of the button you are interested in. Configure each operator button here.

```
private void configureOperatorButton()
{
    configureOperatorButtonWithId( R.id.button_plus );
    configureOperatorButtonWithId( R.id.button_minus );
    configureOperatorButtonWithId( R.id.button_multiply );
    configureOperatorButtonWithId( R.id.button_divide );
    configureOperatorButtonWithId( R.id.button_modulo );
}
```

In `configureOperatorWithId()`, use the `layout` field to get a reference to the id in question. Associate a click listener with `setOnClickListener()`.

```
private void configureOperatorButtonWithId( int id )
{
    layout.findViewById( id )
        .setOnClickListener( createOperatorOnClickListener() );
}
```

Finally, create a helper function called `createOperatorOnClickListener()` and toast the text configured on the button.

```
private View.OnClickListener createOperatorOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity() ,
                ( (Button) v ).getText() ,
                Toast.LENGTH_SHORT )
                .show();
        }
    };
}
```

Run your tests, they should pass.

When you run the application, all the operator buttons should toast when you click on them.

Commit your work.

7.7 Equals Button

Finally, we'll add interactivity to the equals button.

Promote the local variable `equalsButton` to a field. Add a test that clicks on the button and verifies the toasted text.

```
@Test
public void equalsButtonShouldToast() throws Exception
{
    equalsButton.performClick();
    assertThat( ShadowToast.getTextOfLatestToast(),
                equalTo( equalsButton.getText() ) );
}
```

In `ButtonsFragment`, configure the click listener. Create a function called `configureEquals()` and call from `onCreate()`.

```
private void configureEqualsButton()
{
    layout.findViewById( R.id.button_equals )
        .setOnClickListener( new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity(),
                            ((Button) v).getText(),
                            Toast.LENGTH_SHORT )
                .show();
        }
    } );
}
```

Run all the tests.

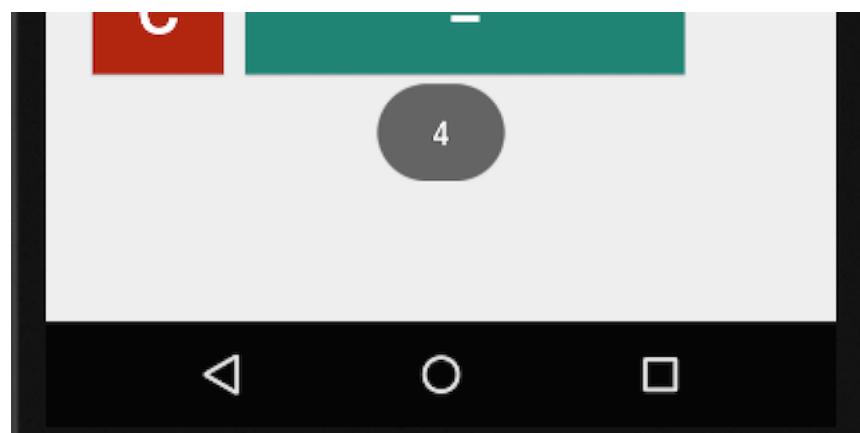
Run the application, all of your buttons should respond to clicks.

Commit when the test pass and the buttons respond.

Summary

In this chapter, we added interactivity to our application and laid the foundation for future chapters.

We added click listeners to all buttons in the application. Each button reports the text that is configured on it when pressed. This text will be helpful in future chapters.



In [Chapter 8](#), we will update the display when numbers, operators, or the clear button are pressed.

In [Chapter 9](#) we will add the full logic to manage operations and numbers, including the equals button.

The code related to this chapter follows.

ButtonsFragmentTest

```
@RunWith(RobolectricGradleTestRunner.class)

public class ButtonsFragmentTest
{
    private ButtonsFragment fragment;

    private Button button1;
    private Button button2;
    private Button button3;
    private Button button4;
    private Button button5;
    private Button button6;
    private Button button7;
    private Button button8;
    private Button button9;
    private Button button0;

    private Button modulo;
    private Button divide;
    private Button minus;
    private Button multiply;
    private Button plus;

    @Before
    public void setUp() throws Exception
    {
        fragment = ButtonsFragment.newInstance();
        startFragment( fragment );

        // Numbers
        button1 = getButtonById( R.id.button1 );
        button2 = getButtonById( R.id.button2 );
        button3 = getButtonById( R.id.button3 );
        button4 = getButtonById( R.id.button4 );
        button5 = getButtonById( R.id.button5 );
        button6 = getButtonById( R.id.button6 );
        button7 = getButtonById( R.id.button7 );
        button8 = getButtonById( R.id.button8 );
        button9 = getButtonById( R.id.button9 );
        button0 = getButtonById( R.id.button0 );

        // Operators
        plus = getButtonById( R.id.button_plus );
        minus = getButtonById( R.id.button_minus );
        multiply = getButtonById( R.id.button_multiply );
        divide = getButtonById( R.id.button_divide );
        modulo = getButtonById( R.id.button_modulo );
    }

    // ...
}
```

```

//region Presence Verifications
@Test
public void shouldHaveOneButton() throws Exception
{
    assertViewIsVisible( button1 );
    assertThat( button1.getText().toString(),
                equalTo( getString( R.string.BUTTON_1 ) ) );
}

// Test for each number button
// ...

@Test
public void shouldHavePlusOperator() throws Exception
{
    assertViewIsVisible( plus );
    assertThat( plus.getText().toString(),
                equalTo( getString( R.string.BUTTON_PLUS ) ) );
}

@Test
public void shouldHaveMinusOperator() throws Exception
{
    assertViewIsVisible( minus );
    assertThat( minus.getText().toString(),
                equalTo( getString( R.string.BUTTON_MINUS ) ) );
}

@Test
public void shouldHaveMultiplyOperator() throws Exception
{
    assertViewIsVisible( multiply );
    assertThat( multiply.getText().toString(),
                equalTo( getString( R.string.BUTTON_MULTIPLY ) ) );
}

@Test
public void shouldHaveDivideOperator() throws Exception
{
    assertViewIsVisible( divide );
    assertThat( divide.getText().toString(),
                equalTo( getString( R.string.BUTTON_DIVIDE ) ) );
}

@Test
public void shouldHaveModuloOperator() throws Exception
{
    assertViewIsVisible( modulo );
    assertThat( modulo.getText().toString(),
                equalTo( getString( R.string.BUTTON_MODULO ) ) );
}
//endregion

```

```

//region Number Toast Verifications
@Test
public void oneButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button1 );
}

@Test
public void twoButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button2 );
}

@Test
public void threeButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button3 );
}

@Test
public void fourButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button4 );
}

// More number tests here
// ...

@Test
public void eightButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button8 );
}

@Test
public void nineButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button9 );
}

@Test
public void zeroButtonShouldToast() throws Exception
{
    verifyToastAfterButtonClick( button0 );
}

void verifyToastAfterButtonClick( Button button )
{
    button.performClick();
    assertThat( ShadowToast.getTextOfLatestToast(),
                equalTo( button.getText() ) );
}
//endregion

```

```

//region Operator Toast Verifications
@Test
public void plusButtonShouldToast() throws Exception
{
    verifyToastAfterOperatorClick( plus );
}

@Test
public void minusButtonShouldToast() throws Exception
{
    verifyToastAfterOperatorClick( minus );
}

@Test
public void multiplyButtonShouldToast() throws Exception
{
    verifyToastAfterOperatorClick( multiply );
}

@Test
public void divideButtonShouldToast() throws Exception
{
    verifyToastAfterOperatorClick( divide );
}

@Test
public void moduloButtonShouldToast() throws Exception
{
    verifyToastAfterOperatorClick( modulo );
}

void verifyToastAfterOperatorClick( Button button )
{
    button.performClick();
    assertThat( ShadowToast.getTextOfLatestToast(),
                equalTo( button.getText() ) );
}
//endregion

// Helper Functions
// ...

```

ButtonsFragment

```
public class ButtonsFragment extends Fragment
{
    private View layout;

    public ButtonsFragment()
    {
    }

    public static ButtonsFragment newInstance()
    {
        return new ButtonsFragment();
    }

    @Override
    public View onCreateView( LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState )
    {
        layout = inflater.inflate( R.layout.fragment_buttons, container, false );

        configureNumberButtons();
        configureOperatorButton();
        configureEqualsButton();

        return layout;
    }

    private void configureNumberButtons()
    {
        configureNumberButtonWithId( R.id.button1 );
        configureNumberButtonWithId( R.id.button2 );
        configureNumberButtonWithId( R.id.button3 );
        configureNumberButtonWithId( R.id.button4 );
        configureNumberButtonWithId( R.id.button5 );
        configureNumberButtonWithId( R.id.button6 );
        configureNumberButtonWithId( R.id.button7 );
        configureNumberButtonWithId( R.id.button8 );
        configureNumberButtonWithId( R.id.button9 );
        configureNumberButtonWithId( R.id.button0 );
    }

    private void configureNumberButtonWithId( int id )
    {
        layout.findViewById( id )
            .setOnClickListener( createNumberOnClickListener() );
    }
}
```

```

private View.OnClickListener createNumberOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity(),
                ((Button) v).getText(),
                Toast.LENGTH_SHORT )
                .show();
        }
    };
}

private void configureOperatorButton()
{
    configureOperatorButtonWithId( R.id.button_plus );
    configureOperatorButtonWithId( R.id.button_minus );
    configureOperatorButtonWithId( R.id.button_multiply );
    configureOperatorButtonWithId( R.id.button_divide );
    configureOperatorButtonWithId( R.id.button_modulo );
}

private void configureOperatorButtonWithId( int id )
{
    layout.findViewById( id )
        .setOnClickListener( createOperatorOnClickListener() );
}

private View.OnClickListener createOperatorOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity(),
                ((Button) v).getText(),
                Toast.LENGTH_SHORT )
                .show();
        }
    };
}

```

```
private void configureEqualsButton()
{
    layout.findViewById( R.id.button_equals )
        .setOnClickListener( new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity(),
                ((Button) v).getText(),
                Toast.LENGTH_SHORT )
                .show();
        }
    });
}
```

Chapter 8

Updating the Display

In this chapter, we will update the display based on button clicks.

You should strive for low [coupling](#) in your applications. This means your components shouldn't rely on the implementation of other components. In our case, we don't want the fragments talking directly to each other.

We use a [system bus](#) as an intermediary. The bus will publish calculator events and components will react.

Note: You could use the [observer/listener](#) pattern, but I prefer to use a bus. In this pattern, the intermediary would be the activity.

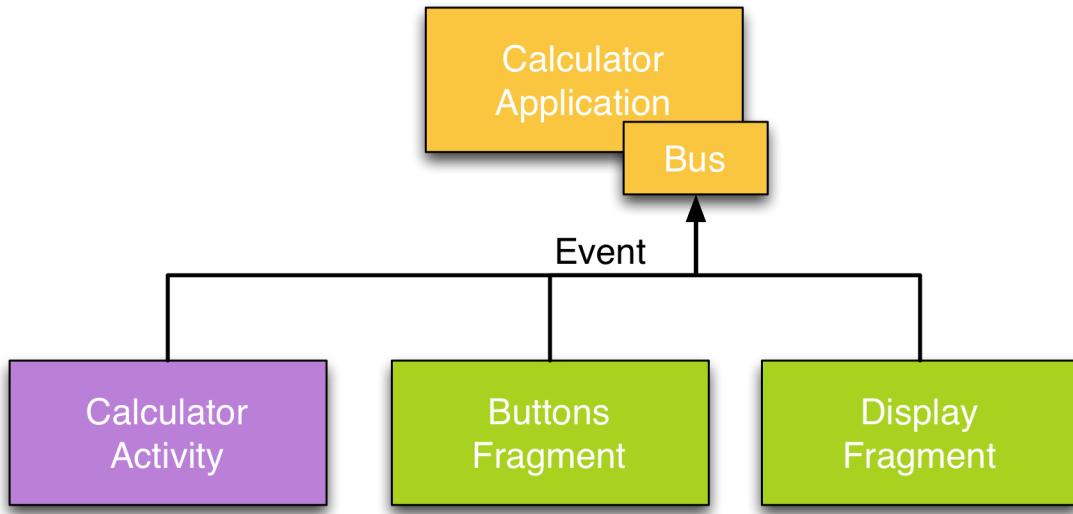
In this chapter we will:

1. Create an application class to host the bus.
2. Integrate an event bus.
3. Post events for button presses.
4. Update the display based on events.

We'll flesh out more of the logic in [Chapter 9](#).

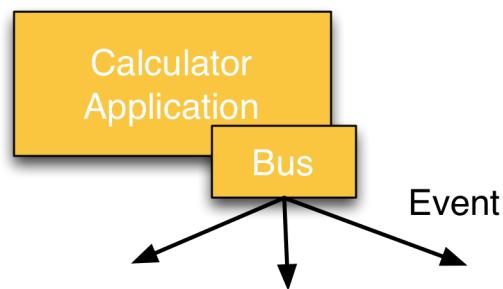
8.1 Routing Calculator Events

When a user clicks a button, we'd like the display to update. In our system, `ButtonsFragment` produces events and the `DisplayFragment` reacts.



All events posted to the bus will be handled by the `CalculatorApplication`.

After an event is posted to the bus, it is broadcast to the rest of the system.



Components register with the bus and then subscribe to interesting events. When the event happens, they have the opportunity to react to the event.¹

For example, let's say you press a number button. Here's the series of interactions that would occur:

1. `ButtonsFragment` posts a `NumberEvent`.
2. `CalculatorActivity`'s bus broadcasts the event.
3. `DisplayFragment` registers with the bus and subscribes to the `NumberEvents`.
The display updates with the number pressed.

We'll work through the foundations and then make each button work as expected.

¹For one broadcast, depending on your set up, several components may react to a single event.

8.2 Add Application & Test

The first step to setting up our event bus is to create an **Application** class. This will be the clearing house and nerve center for the event bus since it's always running while our application is operating.

Application Test

Start by creating the test file in the `robolectric_tests` module. Create a file called `CalculatorApplicationTest.java` in the `com>greenlifesoftware>calculator` package.

Add the test runner and sanity check (non-**null** test) to your test file.

```
@RunWith(RobolectricGradleTestRunner.class)

public class CalculatorApplicationTest
{
    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( CalculatorApplication.getInstance() );
    }
}
```

Use the IDE shortcut  to import relevant classes, create the `CalculatorApplication.java` class in the `app` module.

After the class exists, create the function `getInstance()`.

Application Singleton

In `CalculatorApplication.java`, extend the `Application` class, otherwise you'll see a `ClassCastException` when you run the tests.

```
public class CalculatorApplication extends Application
{
    private static CalculatorApplication instance = new CalculatorApplication();

    public static CalculatorApplication getInstance()
    {
        return instance;
    }
}
```

It's best practice to set your application up as a `singleton`. To do this, have the `getInstance()` function return a private instance of your application.

Making `getInstance()` a `static` function means that you can call `CalculatorActivity.getInstance()` from anywhere in the application.

Add to Manifest

To use the application class, we'll add the `name` attribute to the `<application>` tag in `AndroidManifest.xml`.

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme"
    android:name=".CalculatorApplication">
```

Note: Using `.` in front of the class name is shorthand for the full package name `com.greenlifesoftware.calculator`.

Now, when the application starts it will start your application automatically.

Run the tests and when they pass, commit your changes.

8.3 Setup Event Bus

We will be using [Otto](#) as our event bus.² Otto deals in events – some components post and others subscribe. You create your own event types to represent events in the system.

A component (e.g. an [Activity](#) or [Fragment](#)) posts an event to the bus with an optional message. Other components in the system register for events that are interesting to them and react to events.

Integrate Otto

Add the dependency to your `build.gradle` file in the `app` module so that we can reference throughout our project.

```
dependencies {
    ...
    compile 'com.squareup:otto:1.3.5'
}
```

After updating the dependency, sync your gradle project.

Bus Test

Add a test to `CalculatorApplicationTest.java` and use an [accessor](#) to get a copy of the bus called `getBus()`. Use the IDE to generate the function in your application class.

```
@Test
public void shouldHaveBus() throws Exception
{
    assertNotNull( application.getBus() );
}
```

²Square has created many useful [open source libraries](#) for Android.

Application Bus

Now we'll add the **bus** field to `CalculatorApplication.java`. We already created a getter in the last section. Update it to return **bus**. Finally, initialize the bus in the `onCreate()` lifecycle method.

```
private Bus bus;

public Bus getBus()
{
    return bus;
}

@Override
public void onCreate()
{
    super.onCreate();
    instance.bus = new Bus();
}
```

Add the bus to the application's `instance` variable.

After adding these classes, the tests should still pass.

Commit your work.

8.4 Number Events

In this section, we will create the plumbing that allows us to post events to the bus in the [Post Number Event](#) section.

Update Button Click Test

Open  `ButtonsFragmentTest.java` and navigate your cursor to the test named `verifyToastAfterButtonClick()`.

You can rename all usages of variables, functions, and classes (even those in other files) by using  +  when your cursor is on the name you want to change.³ Update the name to `verifyNumberEvent()`.

Replace the contents of the function with the lines below.

```
void verifyNumberEvent( Button button )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof NumberEvent );
    assertThat( ((NumberEvent) event).getNumber(),
               equalTo( button.getText() ) );
}
```

First, use the button to fire the event. Then retrieve the last event fired and ensure it's the type we expect. Finally, ensure that the expected text is bundled with the event.

There's a lot of red here. We'll work through it step by step in this section.

³On a Mac, you may need to use the  button.

Create Bus Helper

In our newly updated test, we reference a field called **busHelper**, which doesn't exist. Create the field in our `ButtonsFragmentTest.java`.

```
BusHelper busHelper;
```

You can use the shortcut  to generate the field when your cursor is on **busHelper** in the verification test. You can then use this same technique to create the `BusHelper.java` file from the field declaration.

If you do it manually, create a class named `BusHelper.java` in the `support` directory.

Next, add a function `getLastEvent()` and return a new `BaseEvent`.

```
public class BusHelper
{
    public BaseEvent getLastEvent()
    {
        return new BaseEvent();
}
```

Note: We're doing the next, simplest thing to get our tests to pass, or in this case, compile. We'll extend the function later when we add more tests that force us to flesh it out properly.

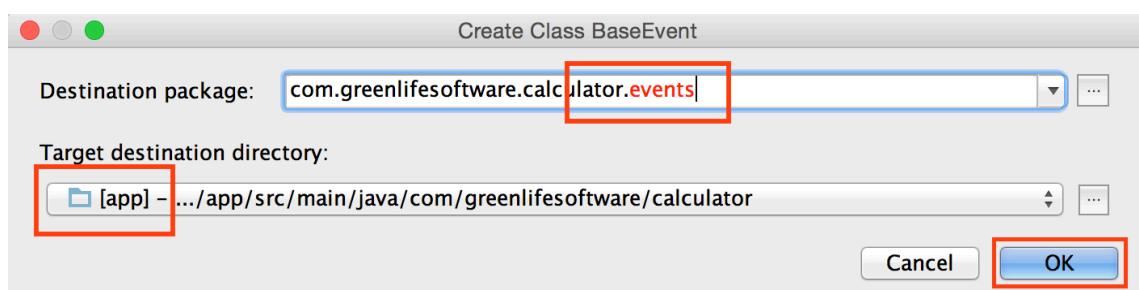
Finally, initialize the **busHelper** field in the test's `setUp()`.

```
@Before
public void setUp() throws Exception
{
    ...
    busHelper = new BusHelper();
}
```

Create Base Event

Create a generic event type, named `BaseEvent.java`, that will be used to post to the bus (we will post events later in the [Post Number Event](#) section). Create a package called `events` to hold your event types since there will be several by the time we're done.

You can use the IDE to generate this class and package at the same time. In the `ButtonsFragmentTest.java` file, navigate to the usage of `BaseEvent` and use  to generate the class for you.



In the `Create Class BaseEvent` dialog, simply add the word “events” to the end of the package name. Make sure to change the module to `app`.

Click `OK`.

Our event class is simple.

```
public class BaseEvent
{
}
```

When we build specific event types, they will extend this `BaseEvent`. These events can be simple like this or you can add data to them.

Note: We won’t be creating tests for this class since it has no functionality. When writing functionality beyond getters, setters, and constructors, a test should be written.

Create Number Event

Now we'll create a number event that will be used to inform the rest of the calculator that a number was pressed.

Create the `NumberEvent.java` class and extend `BaseEvent`. Put it in the `events` package.

```
public class NumberEvent extends BaseEvent
{
}
```

Finally, add a getter called `getNumber()` so that we can retrieve the number when responding to the event.

```
public String getNumber()
{
    return number;
}
```

Note: It makes sense use a `String` type since we can get that from the button and don't need additional infrastructure.

Our build errors are gone, but our tests aren't passing. That's because we need to post events to the bus and to verify which events were fired.

Rename Toast Tests

When we updated how the tests are verified, we didn't update the names of the tests to match the new reality. The operator and number button tests all reference "Toast" in their name (e.g. `oneButtonShouldToast()`).

Open the IDE's search and replace feature with the $\text{⌘} + \text{R}$ shortcut. Search for the word "Toast" and replace it with "PostEvent".



You can use `Replace All` to replace them all at once.

If you prefer to step through each one, use `Replace`. Once you replace one usage, it jumps to the next.

Your final name should look like `oneButtonShouldPostEvent()`.

8.5 Post Number Event

In this section, we add the ability to post events to the application bus and the ability to verify which events were posted.

The assertions we added to the test helper in the [Update Button Click Test](#) section will pass again by the end of this section.

We verified the event was fired by using a subscription to a particular event type. We'll use this subscription model to update the display when a calculator button is pressed.

Update Click Listener

We need a way to communicate which button was pressed. A natural place to add this functionality is in the button clicking code.

To do this, open `ButtonsFragment.java` and replace the toasting code in `createNumberOnClickListener()`'s `onClick()` with code that posts to the application's bus.

```
@Override  
public void onClick( View v )  
{  
    String number = ((Button) v).getText().toString();  
    CalculatorApplication.postToBus( new NumberEvent( number ) );  
}
```

Note: Both lines we added caused build errors. In the following sections, we use IDE shortcuts to create the `postToBus()` function and the constructor in `NumberEvent.java`.

Create a variable `number` to store the string configured on the button (e.g. “1”). We used this value in the toast and now we'll use it for constructing the `NumberEvent`.

Finally, we post the event to the bus.

Update Number Event

Add a constructor to `NumberEvent.java` that takes a **String**. This will be used to associate the number with the event.

```
public NumberEvent( String number )
{
    this.number = number;
}
```

Inside the constructor, save the argument value to the **number** field.

Post to Bus

We need a convenient way to post any type of event to the bus from any activity or fragment in our application.

To do this, open `CalculatorApplication.java` and add a function named **postToBus()**. Use a **BaseEvent** as the argument type so we can handle any event type.

```
public static void postToBus( BaseEvent event )
{
    getInstance().getBus().post( event );
}
```

Inside the function, use the bus instance and Otto's **post()** to fire the event.

Bus Helper

Now that we're posting the event, we need a way to verify that it occurred. To capture these events, we'll extend our helper class (defined in the [Create Bus Helper](#) section) that we can register with the system bus.

More than one event may occur during a test, so we'll create an array named **events** to store them. Using **BaseEvent** as the type allows us to capture all the events that flow through the system.

```
public class BusHelper
{
    private ArrayList<BaseEvent> events = new ArrayList<>();

    public BaseEvent getLastEvent()
    {
        if (!events.isEmpty())
        {
            return events.get( events.size() - 1 );
        }

        return null;
    }

    @Subscribe
    public void onAnyEvent( BaseEvent event )
    {
        events.add( event );
    }
}
```

Update **getLastEvent ()** to return the last published event (last array item).⁴

Finally, add a subscription to receive system events. Use the **@Subscribe** annotation and add received events to the back of the **events** array using **add ()**.

Note: The name of the function doesn't really matter. Otto uses the event type to react to events.

⁴It's wise to check the size of your array, so we've added a non-empty check.

Test Registration

Finally, to receive event notification in our tests, register the `busHelper` in `ButtonsFragmentTest.java`'s `setUp()` function.

```
@Before  
public void setUp() throws Exception  
{  
    ...  
    busHelper = new BusHelper();  
    CalculatorApplication.getInstance().getBus().register( busHelper );  
}
```

Run the tests, they should pass.

If your tests don't pass, check the following:

1. You are using `performClick()` to kick off the event ([Update Button Click Test](#)).
2. You are posting the event to the bus ([Update Click Listener](#)).
3. You are subscribing to `BaseEvents` in `BusHelper` ([Bus Helper](#)).
4. You are storing the last event in `BusHelper` ([Bus Helper](#)).
5. You are returning the last event properly from `BusHelper` ([Bus Helper](#)).
6. You subscribed for event notifications in your test ([Test Registration](#)).

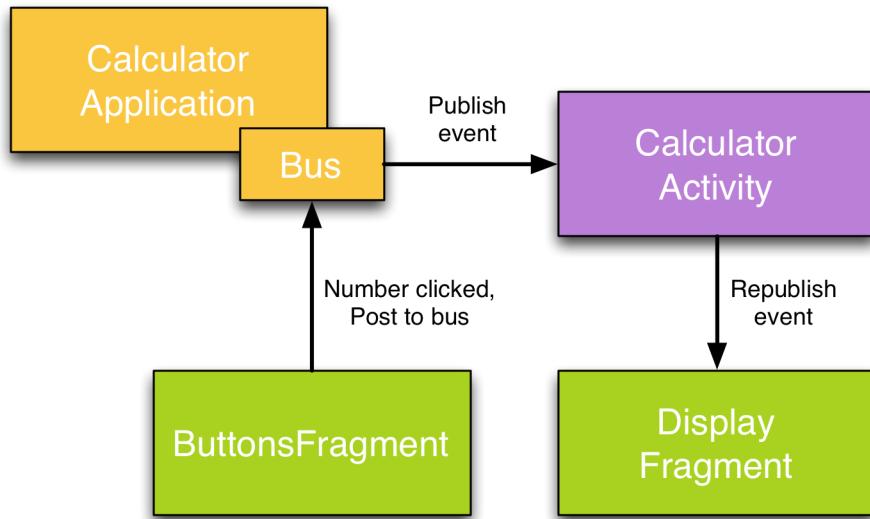
If this checklist doesn't help, check that your events are firing as expected with the debugger. Add breakpoints in the fragment, test code, and bus helper to help you debug the problem.

You should see events triggered by button clicks flow to the application. These events should be stored in `BusHelper.java` after they are published.

8.6 Update Number Display

We've set up most of the infrastructure to post **NumberEvents** to the application, but now we need to communicate these events down to the display.

We'll set up the **DisplayFragment** as a dumb terminal. It accepts commands to update the display, but will not make decisions on how to handle numbers, operators, and other events. We will add that logic later in [Chapter 9](#) when we add state tracking to the calculator.



1. Event is initiated by a button press in **ButtonsFragment**.
2. Event is posted to the bus in **CalculatorApplication**.
3. Otto posts the event to the rest of the system.
4. **CalculatorActivity** subscribes to and listens for **NumberEvents**.
5. **CalculatorActivity** creates and posts a **DisplayEvent**.
6. The **DisplayFragment** subscribes to **DisplayEvent** and updates its view when the event is received.

Display Event Test

It's time to write a failing test in `ButtonsFragmentTest.java` to force us to wire up the rest of the display.

First, create a field `busHelper`, initialize in `setUp()`, and register with the application bus in `CalculatorActivityTest.java`. See sections [Create Bus Helper](#) and [Test Registration](#) for a refresher.

We add a `bus` field so we can post events from our tests.

```
private Bus bus;
private BusHelper busHelper;

@Before
public void setUp() throws Exception
{
    ...
    bus = CalculatorApplication.getInstance().getBus();
    busHelper = new BusHelper();
    bus.register( busHelper );
}
```

Then add a test called `numberEventShouldFireDisplayEvent()` to `CalculatorActivity.java`.

Note: View updates belong in `DisplayFragmentTest.java`.

The `CalculatorActivity` is not responsible for that behavior, so a test for it doesn't belong here. We'd be crossing into integration testing territory.

In the test, post a **NumberEvent** with a test value to the bus and verify the number of events fired as well as a **DisplayEvent** is fired with the correct value.

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

    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof DisplayEvent );
    assertThat( busHelper.numberOfEvents(), equalTo( 2 ) );
    assertThat( ((DisplayEvent) event).getValue(),
                equalTo( NUMBER_VALUE ) );
}
```

From the test, use the shortcut to create the helper **numberOfEvents()** in BusHelper.java. In this function, return the number of events.

```
public int numberOfEvents()
{
    return events.size();
}
```

We use the other build errors to generate code for us in the next section.

Display Event

Create `DisplayEvent.java` in the `events` package of the `app` module using + . Use the shortcut again to create the getter `getValue()`.

The class should extend `BaseEvent`, have a field `value`, and have a getter `getValue()` that returns `value`.

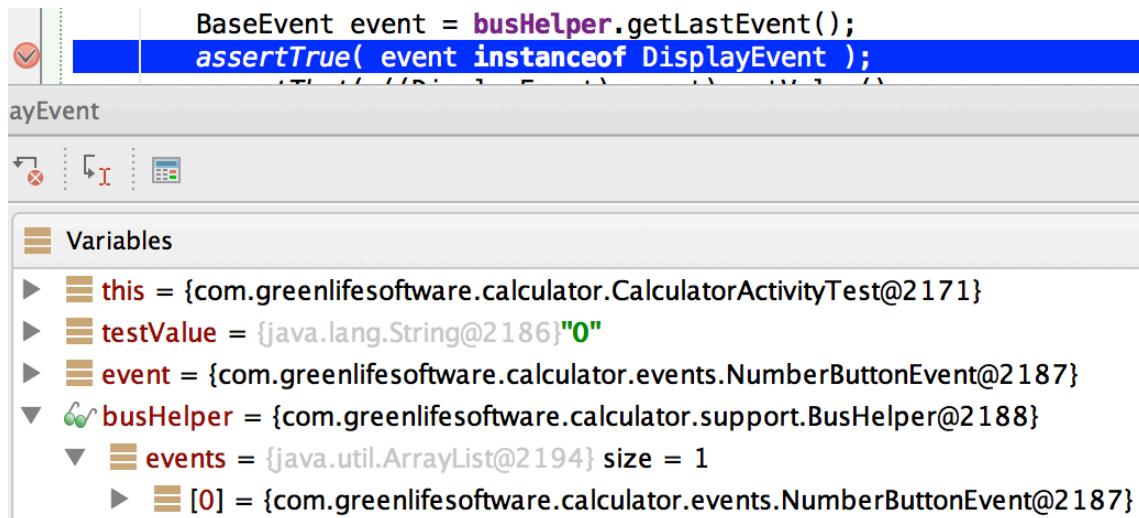
```
package com.greenlifesoftware.calculator.events;

public class DisplayEvent extends BaseEvent
{
    private String value;

    public String getValue()
    {
        return value;
    }
}
```

Now our tests should compile, but fail.

When you debug tests, you'll see that the `NumberEvent` is stored in the `busHelper`'s event list, but there isn't a `DisplayEvent`.



We'll configure the activity to post this event in the following sections.

Register Bus in Activity

In order for `CalculatorActivity` to listen for Otto events, we need to register with the application bus.

Registration is handled in `onResume()`. You can type the method signature manually or use the IDE shortcut `ctrl`+`O` and search for `onResume()`.

```
@Override  
public void onResume()  
{  
    super.onResume();  
    CalculatorApplication.getInstance().getBus().register( this );  
}
```

Note: When overriding lifecycle methods, use `super` to call the parent's version before adding other code to the function. If you don't, you'll see a crash from a `SuperNotCalledException`.

We unregister in `onPause()`:

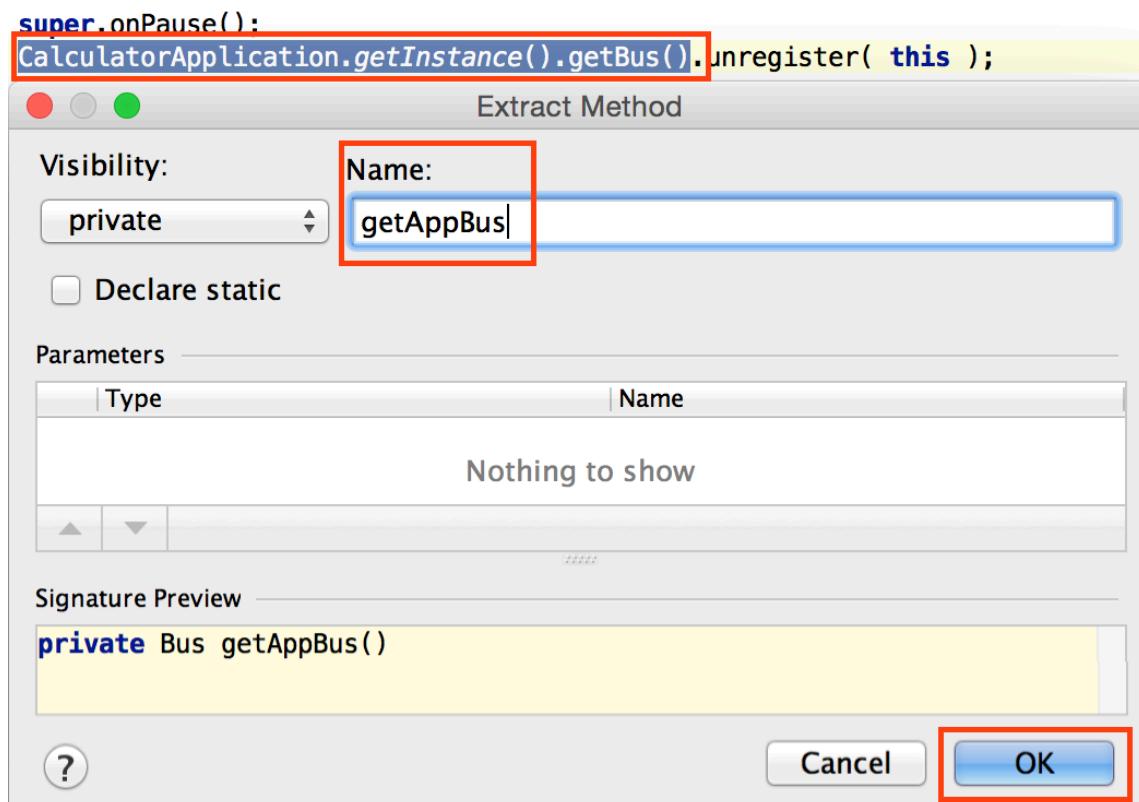
```
@Override  
protected void onPause()  
{  
    super.onPause();  
    CalculatorApplication.getInstance().getBus().unregister( this );  
}
```

There is duplication in the lifecycle methods above:

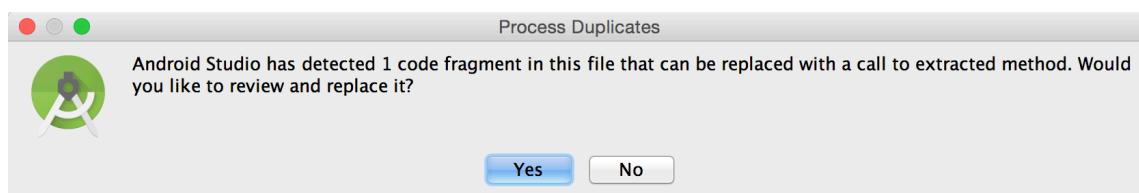
```
CalculatorApplication.getInstance().getBus()
```

We can create a helper function to simplify the code.

The fastest way to create this helper is to select all the way from `CalculatorActivity()` to the last parenthesis of `getBus()` and use the shortcut **⌘ + ⌘ + M** to create a method named `getAppBus()`.



The IDE can automatically replace other instances if you use this shortcut.



Post Display Event

We've registered with the bus, so we can subscribe to **NumberEvents** in **CalculatorActivity.java**.

Using the helper function **postToBus()** in **CalculatorApplication.java**, post a **DisplayEvent**. Pass the number from the **NumberEvent** to the **DisplayEvent**.

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

We'll need to create a constructor in **DisplayEvent.java** to make this happen. You can create it from this file by using the shortcut or manually add it to the file.

```
public DisplayEvent( String value )  
{  
    this.value = value;  
}
```

Run the tests, they pass.

Commit your work.

If we run the application, we won't see the display updating when we press a button. We'll do that next.

Display Fragment Test

Create a test named `shouldUpdateDisplayAfterDisplayEvent()`.

As we did in the [Display Event Test](#) section, add a `bus` field.

Note: We won't need a `busHelper` here since we won't be inspecting events.

In the test, use the `bus` to post a `DisplayEvent`. Then verify that the displayed text is updated to the text passed with the event.

```
@Test
public void shouldUpdateDisplayAfterDisplayEvent() throws Exception
{
    bus.post( new DisplayEvent( TEST_VALUE ) );
    assertThat( display.getText().toString(),
                equalTo( TEST_VALUE ) );
}
```

Create the constant field, `TEST_VALUE`, with + . Set the value to "Test".

```
public static final String TEST_VALUE = "Test";
```

When you run the test, you should see this failure:

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

Now we have a reason to update the display fragment.

Update Display Fragment

First, we register and unregister the bus like we did in the [Register Bus in Activity](#) section.

Next, subscribe to **DisplayEvents**.

```
@Subscribe  
public void onDisplayEvent( DisplayEvent event )  
{  
}
```

Finally, update the displayed text.

```
@Subscribe  
public void onDisplayEvent( DisplayEvent event )  
{  
    EditText display = (EditText) layout.findViewById( R.id.calculator_display );  
    display.setText( event.getValue() );  
}
```

Obtain a reference to the view with id **calculator_display** (defined in the layout file `fragment_display.xml`). Set the display using **setText()** and the text from the **event**.

When you run the tests, they should pass. When you run the application, you should see each press of a number button update the display.

When all the tests pass and your application works, commit your work.

8.7 Update Operator Display

This follows a similar path to the [Post Number Event](#) section.

Update Operator Test

First, add a failing test to `DisplayFragment.java` to ensure the display is updated when an operator event occurs.

```
@Test
public void shouldUpdateDisplayAfterOperatorEvent() throws Exception
{
    String testValue = "%";
    bus.post( new OperatorEvent( testValue ) );
    assertThat( display.getText().toString(),
                equalTo( testValue ) );
}
```

We'll create a new display event type to signify that we've entered an operation state in our application.

Create the `OperatorEvent.java` class and add a constructor that takes a `String`. Store the `operator` passed into the constructor as a field and add a getter.

```
package com.greenlifesoftware.calculator.events;

public class OperatorEvent extends BaseEvent
{
    private final String operator;

    public OperatorEvent( String operator )
    {
        this.operator = operator;
    }

    public String getOperator()
    {
        return operator;
    }
}
```

Update Buttons Fragment

To make the test pass, we'll post an event to the bus. First, we'll add a test for the event to `ButtonsFragmentTest.java`.

Rename⁵ the test helper function `verifyToastAfterOperatorClick()` to `verifyOperatorButtonEvent()`.

Replace the toasting code with verification that the event fired by the button click is an `OperatorEvent` (similar to [Update Button Click Test](#)). Click the button, collect the event, and compare the event type and the value to expected values.

```
private void verifyOperatorButtonEvent( Button button )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof OperatorEvent );
    assertThat( ((OperatorEvent) event).getOperator(),
               equalTo( button.getText() ) );
}
```

Now you should have five failing tests for the operators and one for updating the display.

Open `ButtonsFragment.java`. Update the click listener defined in `createOperatorOnClickListener()` to post an `OperatorEvent` to the bus using the text configured on the button.

```
private View.OnClickListener createOperatorOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            String operator = ((Button) v).getText().toString();
            postToBus( new OperatorEvent( operator ) );
        }
    };
}
```

⁵Use the shortcut  +  to rename all usages.

We can reduce code duplication by creating a helper function called `postToBus()` which will replace usages of `CalculatorApplication.postToBus()`.

```
private void postToBus( BaseEvent event )
{
    CalculatorApplication.postToBus( event );
}
```

At this point, your operator tests should be passing, but your test for updating the display is not.

Update Display for Operator

Finally, register for **OperatorEvents** in **DisplayFragment.java**.

```
@Subscribe  
public void onOperatorDisplay( OperatorEvent event )  
{  
    setDisplay( event.getOperator() );  
}
```

Use  +  to generate **setDisplay()**. The function should take a string to update the display. Here we set the text on the view.

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

Use  +  again to generate **getDisplayView ()**. Return a reference to the display view.

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

Run the tests. They should all pass.

When you run the application, operators should update the display.

Once everything works as expected, commit your work.

Refactor

We're green, so let's clean up `DisplayFragmentTest.java`.

The way it reads now is a little clunky. You want your tests to be self documenting, so let's make the tests read more like a sentence.

```
@Test  
public void shouldUpdateDisplayAfterDisplayEvent() throws Exception  
{  
    postDisplayEvent( TEST_VALUE );  
    assertEquals( TEST_VALUE );  
}
```

This will change most of this class, so make sure to check out the code listing for `DisplayFragmentTest.java` at the end of the chapter to see the end result.

Assertion

If you look at the test assertions, you'll see a similar pattern. They all fetch the displayed text and compare it to some value.

First, create a function called `assertEquals()` and have it take a value for comparison.

```
private void assertEquals( String value )  
{  
    assertThat( display.getText().toString(),  
                equalTo( value ) );  
}
```

Inside the function, assert that the value passed in matches the displayed value.

Update the tests `shouldUpdateDisplayAfterOperatorEvent()`, and `shouldUpdateDisplayAfterDisplayEvent()` to use this helper.

Posting Events

Next, create a helper for posting each event type.

For **DisplayEvents**, create a function named **postDisplayEvent ()**.

```
private void postDisplayEvent( String value )
{
    bus.post( new DisplayEvent( value ) );
}
```

For **OperatorEvents**, create a function named **postOperatorEvent ()**.

```
private void postOperatorEvent( String value )
{
    bus.post( new OperatorEvent( value ) );
}
```

Update the tests to use these functions.

Default Display

Finally, let's add another function named **shouldHaveDefaultDisplay ()**.

```
@Test
public void shouldHaveDefaultDisplay() throws Exception
{
    assertValueDisplayed( getString( R.string.DEFAULT_DISPLAY ) )
}
```

After all these the refactors, your tests should still be passing.

Summary

In this chapter we made all of our buttons (except equals) update the display. Now we're ready to add logic to handle numbers and operators.

■ ButtonsFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.Toast;

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;

public class ButtonsFragment extends Fragment
{
    private View layout;

    public ButtonsFragment()
    {
    }

    public static ButtonsFragment newInstance()
    {
        return new ButtonsFragment();
    }

    @Nullable
    @Override
    public View onCreateView( LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState )
    {
        layout = inflater.inflate( R.layout.fragment_buttons, container, false );

        configureNumberButtons();
        configureOperatorButtons();
        configureEqualsButton();

        return layout;
    }
}
```

```
private void configureNumberButtons()
{
    configureNumberButtonWithId( R.id.button1 );
    configureNumberButtonWithId( R.id.button2 );
    configureNumberButtonWithId( R.id.button3 );
    configureNumberButtonWithId( R.id.button4 );
    configureNumberButtonWithId( R.id.button5 );
    configureNumberButtonWithId( R.id.button6 );
    configureNumberButtonWithId( R.id.button7 );
    configureNumberButtonWithId( R.id.button8 );
    configureNumberButtonWithId( R.id.button9 );
    configureNumberButtonWithId( R.id.button0 );
}

private void configureNumberButtonWithId( int id )
{
    layout.findViewById( id )
        .setOnClickListener( createNumberOnClickListener() );
}

private View.OnClickListener createNumberOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            String number = ( (Button) v ).getText().toString();
            postToBus( new NumberEvent( number ) );
        }
    };
}
```

```
private void configureOperatorButtons()
{
    configureOperatorButtonWithId( R.id.button_plus );
    configureOperatorButtonWithId( R.id.button_minus );
    configureOperatorButtonWithId( R.id.button_multiply );
    configureOperatorButtonWithId( R.id.button_divide );
    configureOperatorButtonWithId( R.id.button_modulo );
}

private void configureOperatorButtonWithId( int id )
{
    layout.findViewById( id )
        .setOnClickListener( createOperatorOnClickListener() );
}

private View.OnClickListener createOperatorOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            String operator = ( (Button) v ).getText().toString();
            postToBus( new OperatorEvent( operator ) );
        }
    };
}
```

```
private void configureEqualsButton()
{
    layout.findViewById( R.id.button_equals )
        .setOnClickListener( new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            Toast.makeText( getActivity(),
                ( (Button) v ).getText(),
                Toast.LENGTH_SHORT )
                .show();
        }
    } );
}

private void postToBus( BaseEvent event )
{
    CalculatorApplication.postToBus( event );
}
}
```

ButtonsFragmentTest.java

```
package com.greenlifesoftware.calculator;

import android.view.View;
import android.widget.Button;

import com.greenlifesoftware.calculator.events.*;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.robolectric.shadows.ShadowToast;

import static com.greenlifesoftware.calculator.support.ResourceLocator.getString;
import static org.hamcrest.core.AreEqual.equalTo;
import static org.junit.Assert.assertNotNull;
import static com.greenlifesoftware.calculator.support.Assert.assertViewIsVisible;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)

public class ButtonsFragmentTest
{
    private ButtonsFragment fragment;

    // Numbers
    private Button button1;
    private Button button2;
    private Button button3;
    private Button button4;
    private Button button5;
    private Button button6;
    private Button button7;
    private Button button8;
    private Button button9;
    private Button button0;

    // Operators
    private Button modulo;
    private Button divide;
    private Button minus;
    private Button multiply;
    private Button plus;

    private BusHelper busHelper;
    private Button equalsButton;
```

```

@Before
public void setUp() throws Exception
{
    fragment = ButtonsFragment.newInstance();
    startFragment( fragment );

    busHelper = new BusHelper();
    CalculatorApplication.getInstance().getBus().register( busHelper );

    // Numbers
    button1 = getButtonById( R.id.button1 );
    button2 = getButtonById( R.id.button2 );
    button3 = getButtonById( R.id.button3 );
    button4 = getButtonById( R.id.button4 );
    button5 = getButtonById( R.id.button5 );
    button6 = getButtonById( R.id.button6 );
    button7 = getButtonById( R.id.button7 );
    button8 = getButtonById( R.id.button8 );
    button9 = getButtonById( R.id.button9 );
    button0 = getButtonById( R.id.button0 );

    // Operators
    plus = getButtonById( R.id.button_plus );
    minus = getButtonById( R.id.button_minus );
    multiply = getButtonById( R.id.button_multiply );
    divide = getButtonById( R.id.button_divide );
    modulo = getButtonById( R.id.button_modulo );

    equalsButton = getButtonById( R.id.button_equals );
}

//region Presence Verifications
@Test
public void shouldNotBeNull() throws Exception
{
    assertNotNull( fragment );
}

@Test
public void shouldHaveButtons() throws Exception
{
    assertViewIsVisible( getViewById( R.id.calculator_buttons ) );
}

@Test
public void shouldHaveOneButton() throws Exception
{
    assertViewIsVisible( button1 );
    assertThat( button1.getText().toString(),
                equalTo( getString( R.string.BUTTON_1 ) ) );
}

```

```

@Test
public void shouldHaveTwoButton() throws Exception
{
    assertViewIsVisible( button2 );
    assertThat( button2.getText().toString(),
                equalTo( getString( R.string.BUTTON_2 ) ) );
}

@Test
public void shouldHaveThreeButton() throws Exception
{
    assertViewIsVisible( button3 );
    assertThat( button3.getText().toString(),
                equalTo( getString( R.string.BUTTON_3 ) ) );
}

@Test
public void shouldHaveFourButton() throws Exception
{
    assertViewIsVisible( button4 );
    assertThat( button4.getText().toString(),
                equalTo( getString( R.string.BUTTON_4 ) ) );
}

@Test
public void shouldHaveFiveButton() throws Exception
{
    assertViewIsVisible( button5 );
    assertThat( button5.getText().toString(),
                equalTo( getString( R.string.BUTTON_5 ) ) );
}

@Test
public void shouldHaveSixButton() throws Exception
{
    assertViewIsVisible( button6 );
    assertThat( button6.getText().toString(),
                equalTo( getString( R.string.BUTTON_6 ) ) );
}

@Test
public void shouldHaveSevenButton() throws Exception
{
    assertViewIsVisible( button7 );
    assertThat( button7.getText().toString(),
                equalTo( getString( R.string.BUTTON_7 ) ) );
}

@Test
public void shouldHaveEightButton() throws Exception
{
    assertViewIsVisible( button8 );
    assertThat( button8.getText().toString(),
                equalTo( getString( R.string.BUTTON_8 ) ) );
}

```

```

@Test
public void shouldHaveNineButton() throws Exception
{
    assertViewIsVisible( button9 );
    assertThat( button9.getText().toString(),
                equalTo( getString( R.string.BUTTON_9 ) ) );
}

@Test
public void shouldHaveZeroButton() throws Exception
{
    assertViewIsVisible( button0 );
    assertThat( button0.getText().toString(),
                equalTo( getString( R.string.BUTTON_0 ) ) );
}

@Test
public void shouldHavePlusOperator() throws Exception
{
    assertViewIsVisible( plus );
    assertThat( plus.getText().toString(),
                equalTo( getString( R.string.BUTTON_PLUS ) ) );
}

@Test
public void shouldHaveMinusOperator() throws Exception
{
    assertViewIsVisible( minus );
    assertThat( minus.getText().toString(),
                equalTo( getString( R.string.BUTTON_MINUS ) ) );
}

@Test
public void shouldHaveMultiplyOperator() throws Exception
{
    assertViewIsVisible( multiply );
    assertThat( multiply.getText().toString(),
                equalTo( getString( R.string.BUTTON_MULTIPLY ) ) );
}

@Test
public void shouldHaveDivideOperator() throws Exception
{
    assertViewIsVisible( divide );
    assertThat( divide.getText().toString(),
                equalTo( getString( R.string.BUTTON_DIVIDE ) ) );
}

@Test
public void shouldHaveModuloOperator() throws Exception
{
    assertViewIsVisible( modulo );
    assertThat( modulo.getText().toString(),
                equalTo( getString( R.string.BUTTON_MODULO ) ) );
}

```

```

@Test
public void shouldHaveEqualsButton() throws Exception
{
    assertViewIsVisible( equalsButton );
    assertThat( equalsButton.getText().toString(),
                equalTo( getString( R.string.BUTTON_EQUALS ) ) );
}

@Test
public void shouldHaveClearButton() throws Exception
{
    Button clearButton = getButtonById( R.id.button_clear );
    assertViewIsVisible( clearButton );
    assertThat( clearButton.getText().toString(),
                equalTo( getString( R.string.BUTTON_CLEAR ) ) );
}

//endregion

//region Number Event Verifications
@Test
public void oneButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button1 );
}

@Test
public void twoButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button2 );
}

@Test
public void threeButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button3 );
}

@Test
public void fourButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button4 );
}

@Test
public void fiveButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button5 );
}

@Test
public void sixButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button6 );
}

```

```

@Test
public void sevenButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button7 );
}

@Test
public void eightButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button8 );
}

@Test
public void nineButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button9 );
}

@Test
public void zeroButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button0 );
}

void verifyNumberEvent( Button button )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof NumberEvent );
    assertThat( ((NumberEvent) event).getNumber(),
               equalTo( button.getText() ) );
}
//endregion

//region Operator Event Verifications
@Test
public void plusButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( plus );
}

@Test
public void minusButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( minus );
}

@Test
public void multiplyButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( multiply );
}

```

```

@Test
public void divideButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( divide );
}

@Test
public void moduloButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( modulo );
}

void verifyOperatorButtonEvent( Button button )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof OperatorEvent );
    assertThat( ((OperatorEvent) event).getOperator(),
               equalTo( button.getText() ) );
}
//endregion

//region Other Verifications
@Test
public void equalsButtonShouldToast() throws Exception
{
    equalsButton.performClick();
    assertThat( ShadowToast.getTextOfLatestToast(),
               equalTo( equalsButton.getText() ) );
}
//endregion

//region Helper Functions
private Button getButtonById( int id )
{
    return (Button) getViewById( id );
}

private View getViewById( int id )
{
    return fragment.getView().findViewById( id );
}
//endregion
}

```

DisplayFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

import com.greenlifesoftware.calculator.events.DisplayEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;

import com.squareup.otto.Bus;
import com.squareup.otto.Subscribe;

public class DisplayFragment extends Fragment
{
    private View layout;

    public DisplayFragment()
    {
    }

    public static DisplayFragment newInstance()
    {
        return new DisplayFragment();
    }

    @Override
    public View onCreateView( LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState )
    {
        layout = inflater.inflate( R.layout.fragment_display, container, false );
        return layout;
    }

    @Override
    public void onResume()
    {
        super.onResume();
        getAppBus().register( this );
    }

    @Override
    public void onPause()
    {
        super.onPause();
        getAppBus().unregister( this );
    }
}
```

```
private Bus getAppBus()
{
    return CalculatorApplication.getInstance().getBus();
}

@Subscribe
public void onDisplayEvent( DisplayEvent event )
{
    setDisplay( event.getValue() );
}

@Subscribe
public void onOperatorDisplay( OperatorEvent event )
{
    setDisplay( event.getOperator() );
}

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

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

DisplayFragmentTest.java

```
package com.greenlifesoftware.calculator;

import android.widget.EditText;

import com.greenlifesoftware.calculator.events.DisplayEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;
import com.squareup.otto.Bus;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static com.greenlifesoftware.calculator.support.Assert.assertViewIsVisible;
import static com.greenlifesoftware.calculator.support.ResourceLocator.getString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertThat;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)
public class DisplayFragmentTest
{
    private static final String TEST_VALUE = "Test";
    private static final String TEST_OPERATOR = "%";

    private DisplayFragment fragment;
    private EditText display;

    private Bus bus;

    @Before
    public void setUp() throws Exception
    {
        fragment = DisplayFragment.newInstance();
        startFragment( fragment );

        display = (EditText) fragment.getView()
                    .findViewById( R.id.calculator_display );

        bus = CalculatorApplication.getInstance().getBus();
    }
}
```

```

@Test
public void shouldNotBeNull() throws Exception
{
    assertNotNull( fragment );
}

@Test
public void shouldHaveDisplay() throws Exception
{
    assertViewIsVisible( display );
}

@Test
public void shouldHaveDefaultDisplay() throws Exception
{
    assertDefaultDisplay();
}

private void assertDefaultDisplay()
{
    assertEquals( getString( R.string.DEFAULT_DISPLAY ) );
}

@Test
public void shouldUpdateDisplayAfterDisplayEvent() throws Exception
{
    postDisplayEvent( TEST_VALUE );
    assertEquals( TEST_VALUE );
}

private void postDisplayEvent( String value )
{
    bus.post( new DisplayEvent( value ) );
}

@Test
public void shouldUpdateDisplayAfterOperatorEvent() throws Exception
{
    postOperatorEvent( TEST_OPERATOR );
    assertEquals( TEST_OPERATOR );
}

private void postOperatorEvent( String value )
{
    bus.post( new OperatorEvent( value ) );
}

private void assertEquals( String value )
{
    assertEquals( display.getText().toString(),
                  value );
}

```

CalculatorActivity.java

```
package com.greenlifesoftware.calculator;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;

import com.greenlifesoftware.calculator.events.DisplayEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.squareup.otto.Bus;
import com.squareup.otto.Subscribe;

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

    @Override
    protected void onResume()
    {
        super.onResume();
        getAppBus().register( this );
    }

    @Override
    protected void onPause()
    {
        super.onPause();
        getAppBus().unregister( this );
    }

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

    private Bus getAppBus()
    {
        return CalculatorApplication.getInstance().getBus();
    }
}
```

CalculatorActivityTest.java

```
package com.greenlifesoftware.calculator;

import android.support.v4.app.Fragment;

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.greenlifesoftware.calculator.events.DisplayEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;
import com.squareup.otto.Bus;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.robolectric.Robolectric;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;

@RunWith(RobolectricGradleTestRunner.class)

public class CalculatorActivityTest
{
    private CalculatorActivity activity;

    private Bus bus;
    private BusHelper busHelper;

    @Before
    public void setUp() throws Exception
    {
        activity = Robolectric.buildActivity( CalculatorActivity.class )
            .create()
            .start()
            .resume()
            .get();

        bus = CalculatorApplication.getInstance().getBus();
        busHelper = new BusHelper();
        bus.register( busHelper );
    }
}
```

```

@Test
public void shouldNotBeNull() throws Exception
{
    assertNotNull( activity );
}

@Test
public void shouldHaveDisplayFragment() throws Exception
{
    assertNotNull( getFragmentById( R.id.display_fragment ) );
}

@Test
public void shouldHaveButtonsFragment() throws Exception
{
    assertNotNull( getFragmentById( R.id.buttons_fragment ) );
}

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

    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof DisplayEvent );
    assertThat( ((DisplayEvent) event).getValue(),
               equalTo( NUMBER_VALUE ) );
}

Fragment getFragmentById( int id )
{
    return activity.getSupportFragmentManager().findFragmentById( id );
}

```

CalculatorApplication.java

```
package com.greenlifesoftware.calculator;

import android.app.Application;

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.squareup.otto.Bus;

public class CalculatorApplication extends Application
{
    private static CalculatorApplication instance = new CalculatorApplication();

    private Bus bus;

    public static CalculatorApplication getInstance()
    {
        return instance;
    }

    public Bus getBus()
    {
        return bus;
    }

    @Override
    public void onCreate()
    {
        super.onCreate();
        instance.bus = new Bus();
    }

    public static void postToBus( BaseEvent event )
    {
        getInstance().getBus().post( event );
    }
}
```

CalculatorApplicationTest.java

```
package com.greenlifesoftware.calculator;

import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertNotNull;

@RunWith(RobolectricGradleTestRunner.class)

public class CalculatorApplicationTest
{

    private CalculatorApplication application;

    @Before
    public void setUp() throws Exception
    {
        application = CalculatorApplication.getInstance();
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( application );
    }

    @Test
    public void shouldHaveBus() throws Exception
    {
        assertNotNull( application.getBus() );
    }
}
```

Event Types

■ BaseEvent.java

```
package com.greenlifesoftware.calculator.events;

public class BaseEvent
{
}
```

■ NumberEvent.java

```
package com.greenlifesoftware.calculator.events;

public class NumberEvent extends BaseEvent
{
    private String number;

    public NumberEvent( String number )
    {
        this.number = number;
    }

    public String getNumber()
    {
        return number;
    }
}
```

DisplayEvent.java

```
package com.greenlifesoftware.calculator.events;

public class DisplayEvent extends BaseEvent
{
    private String value;

    public DisplayEvent( String value )
    {
        this.value = value;
    }

    public String getValue()
    {
        return value;
    }
}
```

OperatorEvent.java

```
package com.greenlifesoftware.calculator.events;

public class OperatorEvent extends BaseEvent
{
    private final String operator;

    public OperatorEvent( String operator )
    {
        this.operator = operator;
    }

    public String getOperator()
    {
        return operator;
    }
}
```

BusHelper.java

```
package com.greenlifesoftware.calculator.support;

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.squareup.otto.Subscribe;

import java.util.ArrayList;

public class BusHelper
{
    private ArrayList<BaseEvent> events = new ArrayList<>();

    public BaseEvent getLastEvent()
    {
        if (!events.isEmpty())
        {
            return events.get( events.size() - 1 );
        }

        return null;
    }

    @Subscribe
    public void onAnyEvent( BaseEvent event )
    {
        events.add( event );
    }

    public int numberOfEvents()
    {
        return events.size();
    }
}
```

build.gradle

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 21
    buildToolsVersion "21.1.2"

    defaultConfig {
        applicationId "com.greenlifesoftware.calculator"
        minSdkVersion 14
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:21.0.3'
    compile 'com.squareup:otto:1.3.5'
}
```

Chapter 9

Calculator State

Our calculator has buttons that update the display, but we have no way to perform calculations or even construct numbers from the sequence of button presses.

In order to do this, we'll need a mechanism to track the state of the calculator.

We don't want to clutter our activity with a bunch of logic, so we'll use a background fragment to track the state.

We lay the groundwork in this chapter and flesh out the state tracking as we work through the next few chapters to add functionality to our calculator.

- In [Chapter 10](#), we construct the numbers entered from key presses.
- In [Chapter 11](#), we clear the calculator display and state.
- In [Chapter 12](#), we handle operators and operands.
- In [Chapter 13](#), we compute the results.

9.1 Adding State

Add a test named `shouldHaveCalculatorStateFragment()` to `CalculatorActivityTest.java` to ensure that the fragment is present.

```
@Test  
public void shouldHaveCalculatorStateFragment() throws Exception  
{  
    Fragment fragment = getStateFragment();  
    assertNotNull( fragment );  
    assertTrue( fragment instanceof CalculatorStateFragment );  
}
```

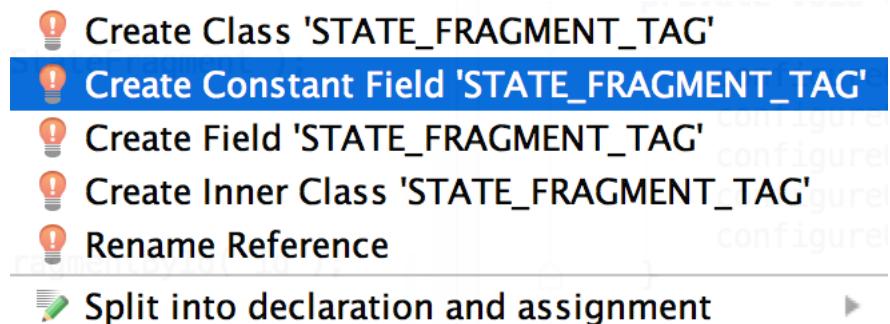
Use the key combination  +  to generate `getStateFragment()`.

```
private Fragment getStateFragment()  
{  
    return activity.getSupportFragmentManager()  
        .findFragmentByTag( CalculatorActivity.STATE_FRAGMENT_TAG );  
}
```

In this function, fetch the fragment manager and use `findFragmentByTag()` to find the background fragment.

Note: You can use a string inline here, but it's better practice to define a constant in `CalculatorActivity.java` to use in the test and activity.

To generate the constant, use the the shortcut  + . From the pop-up, choose **Create Constant Field 'STATE_FRAGMENT_TAG'**.



When the IDE opens the activity, simply select the suggested **String** type by using the  key and enter a value for the string. It really doesn't matter what text you enter here as long as it's unique.

```
public static final String STATE_FRAGMENT_TAG = "CalculatorState";
```

Use  again to exit the generation mode, which is signified by the red boxes around your new constant's value.

In contrast to fragments specified in XML, using a tag is the *only* way to reference a background fragment. When adding the fragment to the fragment manager we can specify a unique string, or tag, that will help us find the **CalculatorStateFragment** later.

Our tests still have a build error – **CalculatorStateFragment** doesn't exist.

9.2 Calculator State Fragment

From `CalculatorActivityTest.java`, use the + shortcut to create `CalculatorStateFragment.java` in the `app` module.

Extend `Fragment` and add an empty constructor.

Override `onCreateView()` using the key combination + .

Since this fragment will only be used in the background to manage the state, it won't have a view. Return `null` from `onCreateView()`.

```
public class CalculatorStateFragment extends Fragment
{
    public CalculatorStateFragment()
    {
    }

    @Override
    public View onCreateView( LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState )
    {
        return null;
    }
}
```

Every class should have a test. Let's add one now.

9.3 Calculator State Fragment Test

From `CalculatorStateFragment.java`, use the key combination + + to generate the test class for you or create it manually in the `robolectric_tests` module.

In `CalculatorStateFragmentTest.java`, add the `@RunWith` annotation to the top of the class.

```
@RunWith(RobolectricGradleTestRunner.class)
```

Use + to generate the sanity check, `shouldNotBeNull()`.

```
@Test
public void shouldNotNull() throws Exception
{
    assertNotNull( fragment );
}
```

Use + to generate the `fragment` field.

```
private CalculatorStateFragment fragment;
```

Generate `setUp()` using + . Inside, use `newInstance()` for fragment instantiation and `startFragment()` to construct it for testing.

```
@Before
public void setUp() throws Exception
{
    fragment = CalculatorStateFragment.newInstance();
    startFragment( fragment );
}
```

Finally, use the shortcut to generate the **newInstance()** function in **CalculatorStateFragment.java**.

Inside the function, return a new instance of the fragment.

```
public static CalculatorStateFragment newInstance()
{
    return new CalculatorStateFragment();
}
```

Run the tests.

```
java.lang.AssertionError
  at org.junit.Assert.fail(Assert.java:86)
  at org.junit.Assert.assertTrue(Assert.java:41)
  at org.junit.Assert.assertNotNull(Assert.java:712)
  ...
...
```

They compile, but fail because we haven't added the fragment to the activity.

9.4 Add State to the Activity

Finally, we'll add the fragment to `CalculatorActivity.java`.

In `onCreate()`, get a reference to the fragment manager by using the `getSupportFragmentManager()` function and start a transaction by using `beginTransaction()`.

Store the returned `transaction` from this call.

```
@Override  
protected void onCreate( Bundle savedInstanceState )  
{  
    super.onCreate( savedInstanceState );  
    setContentView( R.layout.activity_calculator );  
  
    FragmentTransaction transaction = getSupportFragmentManager()  
        .beginTransaction();  
    transaction.add( CalculatorStateFragment.newInstance(),  
        STATE_FRAGMENT_TAG );  
    transaction.commit();  
}
```

Using the `transaction` variable, add the fragment dynamically using `add()`. Give it an instance of the state fragment, created using `newInstance()`, and add a tag for the fragment using the `STATE_FRAGMENT_TAG` constant.¹

Note: Using a uniquely identifiable string allows us to retrieve this fragment later using the fragment manager.

When we're done with the transaction, use `commit()` to start the transaction.²

Run the tests, they should pass.

¹There's [three different ways](#) to add a fragment to the fragment manager.

²According to the [documentation](#), the commit does not happen immediately.

Let's improve the readability of `onCreate()`.

Highlight the fragment transaction code and abstract to a method using the key combination ++. Name it `addStateFragment()`.

```
@Override
protected void onCreate( Bundle savedInstanceState )
{
    super.onCreate( savedInstanceState );
    setContentView( R.layout.activity_calculator );
    addStateFragment();
}

private void addStateFragment()
{
    FragmentTransaction transaction = getSupportFragmentManager()
        .beginTransaction();
    transaction.add( CalculatorStateFragment.newInstance(),
                    STATE_FRAGMENT_TAG );
    transaction.commit();
}
```

Run the tests after the refactor, they should still pass.

9.5 Migrate Activity State

The purpose of `CalculatorStateFragment` is to get the `CalculatorActivity` out of the state tracking business. The activity should only manage the view.

Move Test and Set Up

Move the `numberEventShouldFireDisplayEvent` test from the activity to  `CalculatorStateFragment.java`.

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

    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof DisplayEvent );
    assertThat( ((DisplayEvent) event).getValue(),
               equalTo( NUMBER_VALUE ) );
}
```

Move the `bus` and `busHelper` fields over and initialize them in `setUp()`.

```
private Bus bus;
private BusHelper busHelper;

@Before
public void setUp() throws Exception
{
    fragment = CalculatorStateFragment.newInstance();
    startFragment( fragment );

    bus = CalculatorApplication.getInstance().getBus();
    busHelper = new BusHelper();
    bus.register( busHelper );
}
```

The test would fail if you ran them now.

Move Logic

Move the registration, deregistration, and subscription over from  CalculatorActivity.java to the state fragment.

```
@Override  
public void onResume()  
{  
    super.onResume();  
    getAppBus().register( this );  
}  
  
@Override  
public void onPause()  
{  
    super.onPause();  
    getAppBus().unregister( this );  
}  
  
private Bus getAppBus()  
{  
    return CalculatorApplication.getInstance().getBus();  
}  
  
@Subscribe  
public void onNumberSelected( NumberEvent event )  
{  
    CalculatorApplication.postToBus( new DisplayEvent( event.getNumber() ) );  
}
```

Run the tests, they should pass.

When you run the application, the functionality should be the same. Each number button updates the display, operators are shown, and the clear and equals button toasts.

9.6 Refactor: Base Fragment

Since we're in the green state, it makes sense to simplify the architecture.

You may have noticed a pattern in our fragments. Often they need access to the bus to either post or subscribe to events.

That means we have a lot of duplicated code. Any time you find duplicated code, it's a good idea to refactor.

Create `BaseFragment.java` in the `app` module. This centralized fragment will allow easy access to the bus by providing registration, deregistration, and posting methods.

```
public class BaseFragment extends Fragment
{
    @Override
    public void onResume()
    {
        super.onResume();
        getAppBus().register( this );
    }

    @Override
    public void onPause()
    {
        super.onPause();
        getAppBus().unregister( this );
    }

    protected Bus getAppBus()
    {
        return CalculatorApplication.getInstance().getBus();
    }

    protected void postToBus( BaseEvent event )
    {
        CalculatorApplication.postToBus( event );
    }
}
```

Now update the fragments: `CalculatorStateFragment.java`, `ButtonsFragment.java`, and `DisplayFragment.java`.

Each should extend **BaseFragment**.

Clean up the fragments following the steps below. For each fragment, one or more of the following may apply.

1. Delete the `onPause()`, `onResume()`, `getAppBus()`, and `postToBus()` methods.
2. Replace calls to `CalculatorApplication.postToBus()` with `postToBus()`.
3. Remove unused imports.

Leave subscriptions in the these child fragments alone since each fragment will handle its own events.

Run the tests, they should pass after the refactor.

Summary

In this chapter, we added a fragment to track background state and associated it with our activity. We moved the state tracking logic from the activity into this fragment. Finally, we created a base fragment type and removed code duplication.

We will extend this foundation in future chapters.

calculatorStateFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import com.greenlifesoftware.calculator.events.NumberEvent;
import com.squareup.otto.Subscribe;

public class CalculatorStateFragment extends BaseFragment
{
    public CalculatorStateFragment() {}

    public static CalculatorStateFragment newInstance()
    {
        return new CalculatorStateFragment();
    }

    @Nullable
    @Override
    public View onCreateView( LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState )
    {
        return null;
    }

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

CalculatorStateFragmentTest.java

```
package com.greenlifesoftware.calculator;

import com.greenlifesoftware.calculator.events.*;
import com.greenlifesoftware.calculator.support.*;
import com.squareup.otto.Bus;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static junit.framework.Assert.assertTrue;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.*;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)
public class CalculatorStateFragmentTest
{
    private CalculatorStateFragment fragment;

    private Bus bus;
    private BusHelper busHelper;

    @Before
    public void setUp() throws Exception
    {
        fragment = CalculatorStateFragment.newInstance();
        startFragment( fragment );

        bus = CalculatorApplication.getInstance().getBus();
        busHelper = new BusHelper();
        bus.register( busHelper );
    }

    @Test
    public void shouldNotNull() throws Exception
    {
        assertNotNull( fragment );
    }

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

        BaseEvent event = busHelper.getLastEvent();
        assertTrue( event instanceof DisplayEvent );
        assertThat( ((DisplayEvent) event).getValue(),
                   equalTo( NUMBER_VALUE ) );
    }
}
```

CalculatorActivity.java

```
package com.greenlifesoftware.calculator;

import android.support.v4.app.FragmentTransaction;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;

import com.greenlifesoftware.calculator.events.DisplayEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.squareup.otto.Bus;
import com.squareup.otto.Subscribe;

public class CalculatorActivity extends ActionBarActivity
{
    public static final String STATE_FRAGMENT_TAG = "CalculatorState";

    @Override
    protected void onCreate( Bundle savedInstanceState )
    {
        super.onCreate( savedInstanceState );
        setContentView( R.layout.activity_calculator );
        addStateFragment();
    }

    private void addStateFragment()
    {
        FragmentTransaction transaction = getSupportFragmentManager()
            .beginTransaction();
        transaction.add( CalculatorStateFragment.newInstance(),
            STATE_FRAGMENT_TAG );
        transaction.commit();
    }
}
```

CalculatorActivityTest.java

```
package com.greenlifesoftware.calculator;

import android.support.v4.app.Fragment;

import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.robolectric.Robolectric;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

@RunWith(RobolectricGradleTestRunner.class)

public class CalculatorActivityTest
{
    private CalculatorActivity activity;

    @Before
    public void setUp() throws Exception
    {
        activity = Robolectric.buildActivity( CalculatorActivity.class )
            .create()
            .start()
            .resume()
            .get();
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( activity );
    }

    @Test
    public void shouldHaveDisplayFragment() throws Exception
    {
        assertNotNull( getFragmentById( R.id.display_fragment ) );
    }

    @Test
    public void shouldHaveButtonsFragment() throws Exception
    {
        assertNotNull( getFragmentById( R.id.buttons_fragment ) );
    }
}
```

```
@Test
public void shouldHaveCalculatorStateFragment() throws Exception
{
    Fragment fragment = getStateFragment();
    assertNotNull( fragment );
    assertTrue( fragment instanceof CalculatorStateFragment );
}

private Fragment getStateFragment()
{
    return activity.getSupportFragmentManager()
        .findFragmentByTag( CalculatorActivity.STATE_FRAGMENT_TAG );
}

Fragment getFragmentById( int id )
{
    return activity.getSupportFragmentManager().findFragmentById( id );
}
```

BaseFragment.java

```
package com.greenlifesoftware.calculator;

import android.support.v4.app.Fragment;

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.squareup.otto.Bus;

public class BaseFragment extends Fragment
{
    @Override
    public void onResume()
    {
        super.onResume();
        getAppBus().register( this );
    }

    @Override
    public void onPause()
    {
        super.onPause();
        getAppBus().unregister( this );
    }

    private Bus getAppBus()
    {
        return CalculatorApplication.getInstance().getBus();
    }

    protected void postToBus( BaseEvent event )
    {
        CalculatorApplication.postToBus( event );
    }
}
```

Chapter 10

Constructing Numbers

In this chapter, we'll 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 );
    assertThat( ((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 );
    assertThat( ((AppendEvent) event).getValue(),
               equalTo( 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.

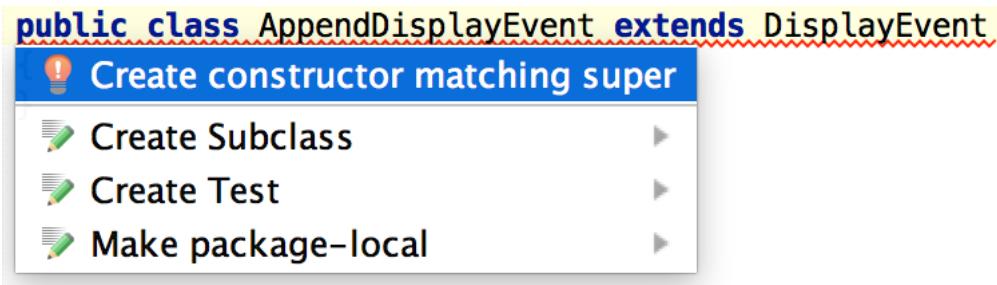
Append Display Event

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 it. 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.

Double Trouble

Run the tests, they are still failing with the same error.

Note: Tests occasionally fail when you run them. There's no guarantee which order the subscriptions will be processed.

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

The reason the tests fail is because we are posting an **AppendEvent**, which is also a **DisplayEvent** (through inheritance). That means both subscriptions are activated when an **AppendEvent** is received.

Note: You can see this happening for yourself if you place breakpoints in all the subscriptions and in the failing test.

To make the test pass, delete the **onDisplayEvent ()** subscription in **DisplayFragment.java**.

Run the tests again – another failure?

More Failures

After removing the subscription for display events, we have a failing test in  `DisplayFragmentTest.java`.

The test `shouldUpdateDisplayAfterDisplayEvent()` no longer applies in its current format. The intention of the test was that a blank display should be updated when a number is pressed.

Let's update the test to match the intention.

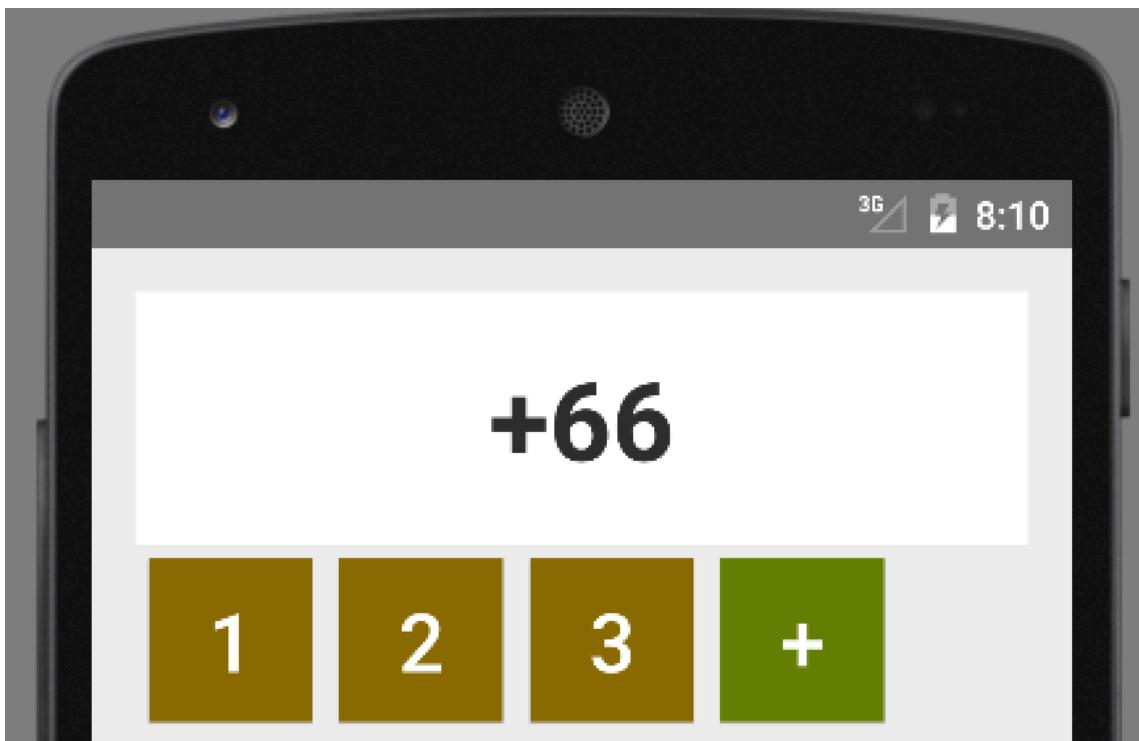
Rename the test to `shouldUpdateBlankDisplayAfterAppendEvent()` and send an `AppendEvent` instead of a `DisplayEvent`.

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

Rerun the tests. Everything should pass.

10.3 Set Display After Operator

Currently if you type a few numbers and then enter an operator, when you type a number after that, it appends it to the display instead of setting the display to the beginning of the new number.



Let's fix that.

Calculator State Test

We have most of the infrastructure required to test this quickly. Add a test called `shouldSetDisplayAfterOperatorEvent()` in `CalculatorStateFragmentTest`.

In the test, post a `NumberEvent`, `OperatorEvent`, and `NumberEvent` to simulate this flow.

We assert that the last event sent was a `SetDisplayEvent`, which tells the display to reset and start a new number. The value in this event should be the value sent last.

```
@Test
public void shouldSetDisplayAfterOperatorEvent() throws Exception
{
    postNumberEvent();
    postOperatorEvent();
    postNumberEvent();

    BaseEvent lastEvent = busHelper.getLastEvent();
    assertTrue( lastEvent instanceof SetDisplayEvent );
    assertThat( ((SetDisplayEvent) lastEvent).getValue(),
               equalTo( NUMBER_VALUE ) );
}
```

Use the shortcut  to create `postOperatorEvent()`. Post an operator event to the bus using a constant value.

```
private void postOperatorEvent()
{
    bus.post( new OperatorEvent( OPERATOR ) );
}
```

Use the shortcut  again to create the test field `OPERATOR`.

```
private static final String OPERATOR = "%";
```

Now we just need to create the new event type.

Create **SetDisplayEvent.java**

We need a new event type to set the display to a particular value (instead of appending).

Use the shortcut  +  from the test file to create the new event type **SetDisplayEvent** in the  events package.

Extend **DisplayEvent**. As we did with the **AppendEvent**, use the  +  shortcut to create the constructor matching super.

```
package com.greenlifesoftware.calculator.events;

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

Now the tests build, but don't pass. We need to handle this event type in  **CalculatorStateFragment.java**.

Update Calculator State

We need a way to track that an operator was pressed and use it when constructing the number. The simplest way to track it is with a boolean value.

When a number is being constructed (contiguous number entry), we should append to the number. However, when an operator was the last event type that we received and a number is pressed, we should set the display to the number value instead of appending.

In `CalculatorStateFragment.java`, add a subscription for `OperatorEvents`. In this subscription, set the value of our boolean field, `operatorWasPressed`, to `true`.

```
private boolean operatorWasPressed = false;

@Subscribe
public void onOperatorEvent(OperatorEvent event)
{
    operatorWasPressed = true;
}
```

Use  +  to create the `operatorWasPressed` field and set it to `false`.

Now we know when an `OperatorEvent` was pressed, but we need to act on this information when handling number events. We also need to clear the flag when an operator was not the last thing entered.

In the `onNumberEvent()` subscription, add an `if` statement using the value of `operatorWasPressed` to either send either a `SetDisplayEvent` when an operator was pressed or an `AppendEvent` when constructing a number.

```
@Subscribe
public void onNumberEvent( NumberEvent event )
{
    if( operatorWasPressed )
    {
        postToBus( new SetDisplayEvent( event.getNumber() ) );
    }
    else
    {
        postToBus( new AppendEvent( event.getNumber() ) );
    }

    operatorWasPressed = false;
}
```

At the end of the subscription, make sure to set `operatorWasPressed` back to `false` since we've left the “operator state” in both cases.

When you run the tests, they pass.

However, the application does not behave as expected. It ignores the first number pressed after an operator because the `DisplayFragment` is not yet responding to `SetDisplayEvents`.

Let's add some display-focused tests to fix this.

Setting the Display

In `DisplayFragmentTest.java`, add a test named `shouldSetDisplayOnSetDisplayEvent()`.

In the test, post an event to ensure that the display value matches `TEST_VALUE`.

```
@Test  
public void shouldSetDisplayOnSetDisplayEvent() throws Exception  
{  
    postSetDisplayEvent();  
    assertThat( display.getText().toString(),  
                equalTo( TEST_VALUE ));  
}
```

Use to create the helper `postSetDisplayEvent()`.

```
private void postSetDisplayEvent()  
{  
    bus.post( new SetDisplayEvent( TEST_VALUE ) );  
}
```

Inside the helper, post a `SetDisplayEvent` to the bus using `TEST_VALUE`.

As expected, the test fails. Now add the subscription for `SetDisplayEvents` to `DisplayFragment.java` to update the display.

```
@Subscribe  
public void onSetDisplay( SetDisplayEvent event )  
{  
    setDisplay( event.getValue() );  
}
```

Run all the tests. Now they pass. Even better, when you run the application, it behaves as expected.

Refactor: Separation of Concerns

It doesn't make sense for the `DisplayFragment` to know anything about the operators in our system. It should be a dumb terminal that only reacts to a limited set of commands: set, clear, and append.

If you poke around `DisplayFragment.java`, you'll see the subscription `onOperatorDisplay()` for handling operator display. Delete this subscription and any associated imports.

Note: We will leverage the `onSetDisplay()` subscription we created in the last section to set the operator value in the display.

Now open `DisplayFragmentTest.java`. Remove the `TEST_OPERATOR` field, the `shouldUpdateDisplayAfterOperatorEvent()` test, and the helper function `postOperatorEvent()`.

New Test, Same Functionality

Since we still want to update the display when an operator is pressed.

Add a new test in `CalculatorStateFragmentTest.java` named `operatorEventShouldFireSetEvent()`.

In the test, post an operator event. Verify the value in the event matches `OPERATOR` and that the event is of type `SetDisplayEvent`.

```
@Test
public void operatorEventShouldFireAppendEvent() throws Exception
{
    postOperatorEvent();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof SetDisplayEvent );
    assertThat( ((SetDisplayEvent) event).getValue(),
               equalTo( OPERATOR ) );
}
```

When you run the test, it fails because it's not firing the right type of event.

Updating Operator Display

Finally, let's update the display.

In `CalculatorStateFragment.java`, post a `SetDisplayEvent` from the `onOperatorSelected()` subscription.

```
@Subscribe  
public void onOperatorSelected( OperatorEvent event )  
{  
    operatorWasPressed = true;  
    postToBus( new SetDisplayEvent( event.getOperator() ) );  
}
```

When you run the tests again, they should pass.

10.4 Limit Digits

We have a limited display width, so let's enforce it with a test. Create a test in `CalculatorStateFragment.java` that constructs a long number and ensures that the operand size doesn't exceed our maximum.

```
@Test  
public void operandShouldNotExceedMaxLength() throws Exception  
{  
    constructTooLongOperand();  
    assertThat( fragment.getOperand().length(),  
                equalTo( CalculatorStateFragment.MAX_OPERAND_LENGTH ) );  
}
```

From the test, use to generate `MAX_OPERAND_LENGTH` as a constant field in `CalculatorStateFragment.java`.

```
public static final int MAX_OPERAND_LENGTH = 10;
```

Note: By prepending `CalculatorStateFragment` to the front of `MAX_OPERAND_LENGTH`, the IDE knows where to create the field. Otherwise, we need to create it manually or generate it in the test file and move it to the fragment. In the test file, use to add an on demand static import for the constant.

Next, generate `constructTooLongOperand()` using . In this function, send more number events than the maximum number of digits.

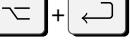
```
private void constructTooLongOperand()  
{  
    for( int counter = 1;  
        counter <= MAX_OPERAND_LENGTH + 2;  
        counter++ )  
    {  
        postNumberEvent();  
    }  
}
```

Finally, use the  shortcut from the test file to generate `getOperand()` in `CalculatorStateFragment.java`. This function will retrieve the current operand for comparisons in our tests.

```
protected String getOperand()
{
    return operand;
}
```

Note: If you select `Create Getter`, the field we create below will be automatically generated for you. If you use the `Create Method 'getOperand()'` option, you will need to generate the `operand` field yourself.

Note: Set the access modifier level to `protected` since we will not need the `operand` outside of a testing perspective.

If not automatically generated, use the shortcut  to generate `operand`. Set the initial value to an empty string.

```
private String operand = "";
```

Now, run the tests for `CalculatorStateFragmentTest.java`.

```
java.lang.AssertionError: Expected: <10> but: was <0>
```

We get this failure because we aren't updating the `operand` after creation.

Limit Operand Digits

To pass the test, we need logic in `CalculatorStateFragment.java` that tracks the current operand.

Since we're handling numbers in the `onNumberEvent()` subscription, it's a natural place to track the `operand`'s value.

First, add an `if` check to the top of the subscription that compares the current length of the operand with the maximum length. Simply `return` from the function after we've reached the maximum length.³

```
if( operand.length() >= MAX_OPERAND_LENGTH )
{
    return;
}
```

We can improve the readability of this conditional by creating a function.

Highlight all the code inside the `if` statement's parenthesis using either the mouse or the key combination $\text{Alt} + \text{Shift} + \text{Up}$. Once highlighted, create a method named `maxOperatorLimitReached()` with $\text{Alt} + \text{Shift} + \text{M}$.

```
if( maxOperatorLimitReached() )
{
    return;
}
```

The function is simple.

```
private boolean maxOperatorLimitReached()
{
    return operand.length() >= MAX_OPERAND_LENGTH;
}
```

³This has the added benefit of not sending unnecessary events to the `DisplayFragment`.

Now it's time to handle `operand`'s value. Since we're using a string to store the value, we can simply append the number from `event` to the current value.

```
operand += event.getNumber();
```

The full subscription looks like this:

```
@Subscribe
public void onNumberSelected( NumberEvent event )
{
    if (maxOperatorLimitReached())
    {
        return;
    }

    if (operatorWasPressed)
    {
        postToBus( new SetDisplayEvent( event.getNumber() ) );
    }
    else
    {
        postToBus( new AppendEvent( event.getNumber() ) );
    }

    operand += event.getNumber();
    operatorWasPressed = false;
}
```

The tests in `CalculatorStateFragmentTest.java` should now pass.

Run the application. We should see the display limited to 10 digits.

Note: In the `Display` style from `styles.xml`, we limited the width with the `maxLength` attribute. This means the `EditText` will never show more than 10 digits. To verify the app works as expected, comment out this line.

Ensuring the Operand's Value

We've checked the length of the value, but we haven't verified that the value is what we expect. Let's add a test to `CalculatorStateFragment.java` named `shouldStoreCorrectMaxOperandValue()`.

```
@Test
public void shouldStoreCorrectMaxOperandValue() throws Exception
{
    constructTooLongOperand();
    assertThat( fragment.getOperand(),
                equalTo( maxTestOperand() ) );
}
```

Call `constructTooLongOperand()`. Compare the value of the longest possible testing operand to the value stored on the operand and use the + shortcut to create the function `maxTestOperand()`.

```
private String maxTestOperand()
{
    String maxOperand = "";

    for (int operandCounter = 1;
        operandCounter <= MAX_OPERAND_LENGTH;
        operandCounter++)
    {
        maxOperand += NUMBER_VALUE;
    }

    return maxOperand;
}
```

This function returns the longest valid number.

Run the tests, they should pass.

10.5 Storing the Operand

We've already added some of the ability to store the value of the operand in the last section. Now we'll make sure it behaves as expected.

Create a test in `CalculatorStateFragmentTest.java`. In the test, `shouldConstructOperand()`, construct a number and compare it to the operand.

```
@Test
public void shouldConstructOperand() throws Exception
{
    String expectedOperand = constructOperand( OPERAND_LENGTH );
    assertThat( fragment.getOperand(),
                equalTo( expectedOperand ) );
}
```

Use the shortcut to create the constant `OPERAND_LENGTH`.

```
private static final int OPERAND_LENGTH = 3;
```

Now use to generate `constructOperand()`.

The simple approach to this function would be to construct an operand of a known size and return a string representing that operand. Instead, we will take a value that specifies the length of the operand to construct. This approach is more flexible and will allow us to use this function when constructing short and long operands.

The parameter **operandLength** represents the size of the operand that we will be constructing. Use this value in your **for** loop conditional.

Inside the **for** loop, post events to the bus and construct the return string.

```
private String constructOperand( int operandLength )
{
    String operand = "";

    for (int operandCounter = 1;
        operandCounter <= operandLength;
        operandCounter++)
    {
        postNumberEvent();
        operand += NUMBER_VALUE;
    }

    return operand;
}
```

Finally, return the constructed **operand** at the end of the function.

Run the tests, they should pass. That's because in the last section, we stored the operand when ensuring it didn't exceed the maximum length.

Refactor: Assertion Duplication

Now that we're in green state, let's look for duplication and clunkiness.

In the last section, we created a function `constructOperand()`. We can use this to simplify `constructTooLongOperand()`. You can replace the `for` loop with one line.

```
private String constructTooLongOperand()
{
    constructOperand( MAX_OPERAND_LENGTH + 2 );
}
```

If we search further, we see there's a few functions that compare the expected operand to the actual value. We can reduce this duplication by creating a centralized function named `assertOperandEquals()`.

This function should take the expected string as input and compare it to the actual value of the operand in `CalculatorStateFragment`.

```
private void assertOperandEquals( String expectedOperand )
{
    assertThat( fragment.getOperand(),
                equalTo( expectedOperand ) );
}
```

We can use this new function to make `shouldConstructOperand()` and `shouldStoreCorrectMaxOperandValue()` more readable.

For example, `shouldConstructOperand()` cleans up like so:

```
@Test
public void shouldConstructOperand() throws Exception
{
    String expectedOperand = constructOperand( OPERAND_LENGTH );
    assertOperandEquals( expectedOperand );
}
```

Ensure the tests still pass.

Refactor: Loop Duplication

You may have noticed that `maxTestOperand()` and `constructOperand()` do basically the same thing. The only difference between the two functions is that one posts an event and one does not. Functions to the rescue!

First, select all the lines in `constructOperand()` and use the shortcut to create a method named `constructOperandMaybePost()`.

After you've abstracted the function, add `true` to the method call to create a build error.

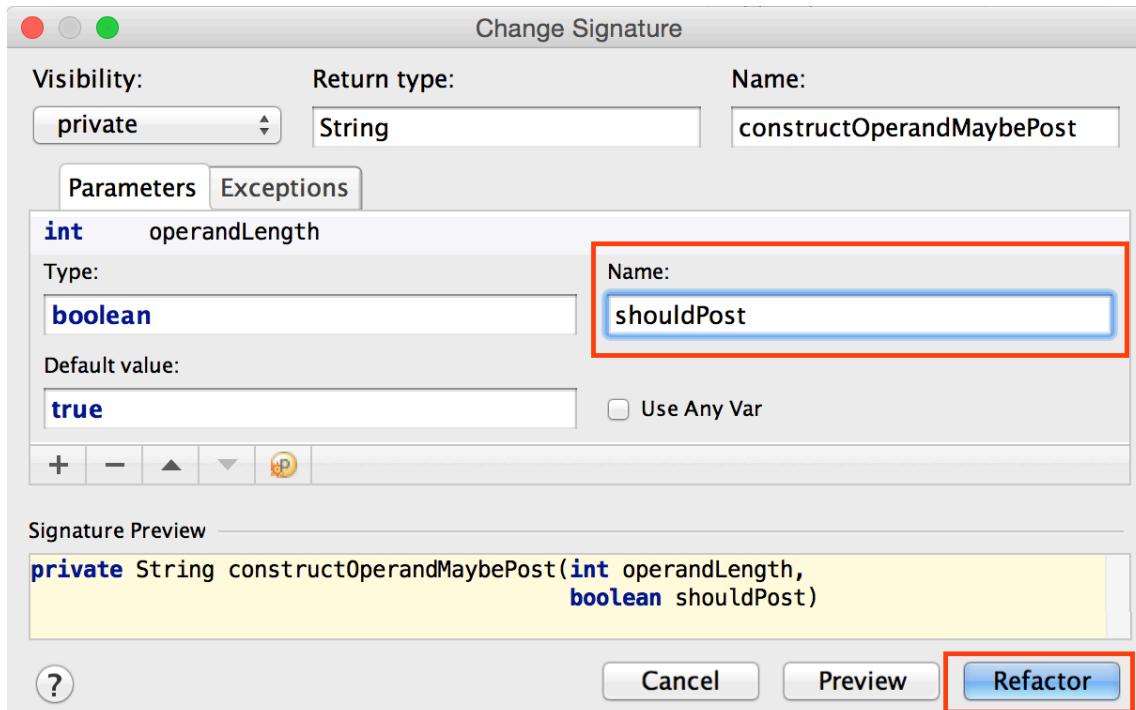
```
private String constructOperand( int operandLength )
{
    return constructOperandMaybePost( operandLength, true );
}
```

You can then immediately solve the build error by using to add a boolean to the signature.

```
private String constructOperand( int operandLength )
{
    return constructOperandMaybePost( operandLength, true );
}
! Add 'boolean' as 2nd parameter to method 'constructOperandMaybePost'
! Create Method 'constructOperandMaybePost' <expectedOperand>
! Remove redundant arguments to call 'constructOperandMaybePost(int)'
```

Select `Add 'boolean' as 2nd parameter to method 'constructOperandMaybePost'` from the pop-up to open a dialog.

In this dialog, you have a lot of options for tweaking the function signature. Click on the new field to change the name to **shouldPost**.



Now that we have a new function and the boolean to tell us whether to post an event, we can set up **constructOperandMaybePost ()**.

The function is largely the same as what `constructOperand()` used to be, but we add an `if` conditional for `shouldPost`. When the value is `true`, we should post an event. If not, we do nothing.

```
private String constructOperandMaybePost( int operandLength,
                                         boolean shouldPost )
{
    String operand = "";

    for (int operandCounter = 1;
         operandCounter <= operandLength;
         operandCounter++)
    {
        if (shouldPost)
        {
            postNumberEvent();
        }

        operand += NUMBER_VALUE;
    }

    return operand;
}
```

Now that we have `constructOperandMaybePost()` set up, we can use it in the `maxTestOperand()` function. We send in the `MAX_OPERAND_LENGTH` for the length and `false` for posting of events.

```
private String maxTestOperand()
{
    return constructOperandMaybePost( MAX_OPERAND_LENGTH, false );
}
```

Ensure the tests still pass.

Refactor: Long Functions

Any time a function gets over four lines of code, I like to refactor. Look for logical groupings of steps that would make sense as an atomic chunk.

Pro tip: Smart usage of functions really ups your code readability.

The IDE makes this task simple. Select the lines you want and use the key combination  +  +  to abstract those lines to a new method.

For example, in `numberEventShouldFireAppendEvent()`, we could move the following lines to a new function named `assertAppendEventWasLast()`:

```
private void assertAppendEventWasLast()
{
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof AppendEvent );
    assertThat( ((AppendEvent) event).getValue(),
               equalTo( NUMBER_VALUE ) );
}
```

In the case of `operatorEventShouldFireAppendEvent()` and `shouldSetDisplayAfterOperatorEvent()`, we want to verify that a `SetDisplayEvent` was the last event sent and that a value matches an expected value.

Create a new function name named `assertSetDisplayEventWithValue()` that takes a `String` value for comparison.

```
private void assertSetDisplayEventWithValue( String value )
{
    BaseEvent lastEvent = busHelper.getLastEvent();
    assertTrue( lastEvent instanceof SetDisplayEvent );
    assertThat( ((SetDisplayEvent) lastEvent).getValue(),
               equalTo( value ) );
}
```

Use this to simplify `numberEventShouldFireAppendEvent()`.

```
@Test
public void operatorEventShouldFireAppendEvent() throws Exception
{
    postOperatorEvent();
    assertSetDisplayEventWithValue( OPERATOR );
}
```

Now simplify `shouldSetDisplayAfterOperatorEvent()`.

```
@Test
public void shouldSetDisplayAfterOperatorEvent() throws Exception
{
    postNumberEvent();
    postOperatorEvent();
    postNumberEvent();
    assertSetDisplayEventWithValue( NUMBER_VALUE );
}
```

Run the tests in `CalculatorStateFragment.java` to ensure they are still passing after these refactors.

10.6 Reset the Operand

When operators are pressed, we don't want to keep constructing the current number. So let's add a test named `shouldResetOperandAfterOperator()` to `CalculatorStateFragment.java` to ensure we restart the construction.

In this test, construct an operand, post an operator event, and ensure that the operand has been reset to the default value (aka empty string).

```
@Test
public void shouldResetOperandAfterOperator() throws Exception
{
    constructOperand( OPERAND_LENGTH );
    postOperatorEvent();
    assertOperandEquals( "" );
}
```

When you run the test, it fails.

```
java.lang.AssertionError:
Expected: ""
      but: was "111"
```

To make the test pass, add logic to `CalculatorStateFragment.java` to reset the value of `operand`. To do this, set the `operand` to an empty string in the `onOperatorSelected()` subscription.

```
@Subscribe
public void onOperatorSelected( OperatorEvent event )
{
    operatorWasPressed = true;
    operand = "";
}
```

Now the tests pass.

Let's add a test named `shouldConstructNumberAfterOperator()` to ensure that we can construct a different number after an operator is entered.

In this function: construct a number, post an operator, then construct a different number, and verify the expected result.

```
@Test
public void shouldConstructNumberAfterOperator() throws Exception
{
    constructOperand( OPERAND_LENGTH );
    postOperatorEvent();

    String expectedOperand = constructOperand( OPERAND_LENGTH + 2 );
    assertOperandEquals( expectedOperand );
}
```

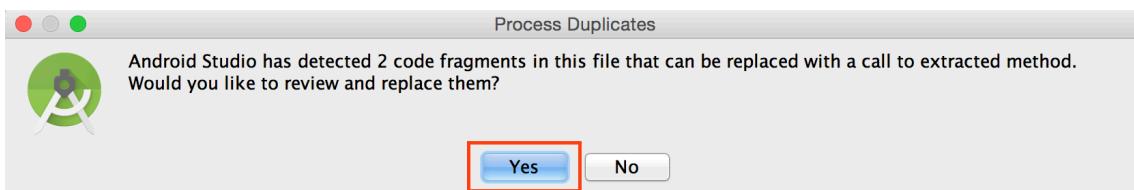
Refactor

We've seen this line of code several times in the tests.

```
constructOperand( OPERAND_LENGTH );
```

Highlight the line using $\text{Esc} + \uparrow$ and create a new method using the shortcut $\text{Alt} + \text{Shift} + \text{C}$. Name it `constructDefaultOperand()`.

Once you do this, it will ask you to replace other usages in the file.



Select `Yes` to review them and `Replace` on each usage to be replaced.

Run the tests, they should pass. If you haven't run the application, check your work to make sure it behaves as expected. Finally, commit your work.

Summary

In this chapter, we handled the logic for constructing a number, or operand, from a series of button presses. We also stored the constructed number for use in future calculations. The next step is to handle operators.

calculatorStateFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import com.greenlifesoftware.calculator.events.AppendEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;
import com.greenlifesoftware.calculator.events.SetDisplayEvent;

import com.squareup.otto.Subscribe;

public class CalculatorStateFragment extends BaseFragment
{
    public static final int MAX_OPERAND_LENGTH = 10;

    private String operand = "";
    private boolean operatorWasPressed = false;

    public CalculatorStateFragment()
    {
    }

    public static CalculatorStateFragment newInstance()
    {
        return new CalculatorStateFragment();
    }

    @Override
    public View onCreateView( LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState )
    {
        return null;
    }
}
```

```

@Subscribe
public void onNumberSelected( NumberEvent event )
{
    if (maxOperatorLimitReached())
    {
        return;
    }

    if (operatorWasPressed)
    {
        postToBus( new SetDisplayEvent( event.getNumber() ) );
    }
    else
    {
        postToBus( new AppendEvent( event.getNumber() ) );
    }

    operand += event.getNumber();
    operatorWasPressed = false;
}

private boolean maxOperatorLimitReached()
{
    return operand.length() >= MAX_OPERAND_LENGTH;
}

@Subscribe
public void onOperatorSelected( OperatorEvent event )
{
    operatorWasPressed = true;
    operand = "";
    postToBus( new SetDisplayEvent( event.getOperator() ) );
}

protected String getOperand()
{
    return operand;
}
}

```

CalculatorStateFragmentTest.java

```
package com.greenlifesoftware.calculator;

import com.greenlifesoftware.calculator.events.AppendEvent;
import com.greenlifesoftware.calculator.events.BaseEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;
import com.greenlifesoftware.calculator.events.SetDisplayEvent;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import com.squareup.otto.Bus;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static com.greenlifesoftware.calculator.CalculatorStateFragment.*;
import static junit.framework.Assert.assertTrue;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.*;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)
public class CalculatorStateFragmentTest
{
    private static final String NUMBER_VALUE = "1";
    private static final String OPERATOR = "%";
    private static final int OPERAND_LENGTH = 3;

    private CalculatorStateFragment fragment;

    private Bus bus;
    private BusHelper busHelper;

    @Before
    public void setUp() throws Exception
    {
        fragment = newInstance();
        startFragment( fragment );

        bus = CalculatorApplication.getInstance().getBus();
        busHelper = new BusHelper();
        bus.register( busHelper );
    }

    @Test
    public void shouldNotNull() throws Exception
    {
        assertNotNull( fragment );
    }
}
```

```

@Test
public void numberEventShouldFireAppendEvent() throws Exception
{
    postNumberEvent();
    assertAppendEventWasLast();
}

private void assertAppendEventWasLast()
{
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof AppendEvent );
    assertThat( ((AppendEvent) event).getValue(),
                equalTo( NUMBER_VALUE ) );
}

@Test
public void operatorEventShouldFireAppendEvent() throws Exception
{
    postOperatorEvent();
    assertSetDisplayEventWithValue( OPERATOR );
}

@Test
public void shouldSetDisplayAfterOperatorEvent() throws Exception
{
    postNumberEvent();
    postOperatorEvent();
    postNumberEvent();
    assertSetDisplayEventWithValue( NUMBER_VALUE );
}

private void assertSetDisplayEventWithValue( String value )
{
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof SetDisplayEvent );
    assertThat( ((SetDisplayEvent) event).getValue(),
                equalTo( value ) );
}

@Test
public void shouldConstructOperand() throws Exception
{
    String expectedOperand = constructDefaultOperand();
    assertOperandEquals( expectedOperand );
}

private String constructDefaultOperand()
{
    return constructOperand( OPERAND_LENGTH );
}

private String constructOperand( int operandLength )
{
    return constructOperandMaybePost( operandLength, true );
}

```

```
private void assertOperandEquals( String expectedOperand )
{
    assertThat( fragment.getOperand() ,
                equalTo( expectedOperand ) );
}
```

```

@Test
public void operandShouldNotExceedMaxLength() throws Exception
{
    constructTooLongOperand();
    assertThat( fragment.getOperand().length(),
                equalTo( MAX_OPERAND_LENGTH ) );
}

@Test
public void shouldStoreCorrectMaxOperandValue() throws Exception
{
    constructTooLongOperand();
    assertOperandEquals( maxTestOperand() );
}

private void constructTooLongOperand()
{
    constructOperand( MAX_OPERAND_LENGTH + 2 );
}

private String maxTestOperand()
{
    return constructOperandMaybePost( MAX_OPERAND_LENGTH, true );
}

@Test
public void shouldResetOperandAfterOperator() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent();
    assertOperandEquals( "" );
}

@Test
public void shouldConstructNumberAfterOperator() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent();

    String expectedOperand = constructOperand( OPERAND_LENGTH + 2 );
    assertOperandEquals( expectedOperand );
}

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

private void postOperatorEvent()
{
    bus.post( new OperatorEvent( OPERATOR ) );
}

```

```
private String constructOperandMaybePost( int operandLength, boolean shouldPost )
{
    String operand = "";

    for (int operandCounter = 1;
        operandCounter <= operandLength;
        operandCounter++)
    {
        if (shouldPost)
        {
            postNumberEvent();
        }

        operand += NUMBER_VALUE;
    }

    return operand;
}
```

DisplayFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

import com.greenlifesoftware.calculator.events.AppendEvent;
import com.greenlifesoftware.calculator.events.ClearEvent;
import com.greenlifesoftware.calculator.events.SetDisplayEvent;

import com.squareup.otto.Subscribe;

public class DisplayFragment extends BaseFragment
{
    private View layout;

    public DisplayFragment()
    {
    }

    public static DisplayFragment newInstance()
    {
        return new DisplayFragment();
    }

    @Override @Nullable
    public View onCreateView( LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState )
    {
        layout = inflater.inflate( R.layout.fragment_display, container, false );
        return layout;
    }

    @Subscribe
    public void onSetDisplay( SetDisplayEvent event )
    {
        setDisplay( event.getValue() );
    }

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

    @Subscribe
    public void onClearDisplay( ClearEvent event )
    }
```

```
{  
    setDisplay( getString( R.string.DEFAULT_DISPLAY ) );  
}  
  
private void setDisplay( String displayString )  
{  
    getDisplayView().setText( displayString );  
}  
  
private String getDisplayString()  
{  
    return getDisplayView().getText().toString();  
}  
  
private EditText getDisplayView()  
{  
    return (EditText) layout.findViewById( R.id.calculator_display );  
}
```

DisplayFragmentTest.java

```
package com.greenlifesoftware.calculator;

import android.widget.EditText;

import com.greenlifesoftware.calculator.events.AppendEvent;
import com.greenlifesoftware.calculator.events.SetDisplayEvent;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import com.squareup.otto.Bus;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static com.greenlifesoftware.calculator.support.Assert.assertViewIsVisible;
import static com.greenlifesoftware.calculator.support.ResourceLocator.getString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertThat;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)
public class DisplayFragmentTest
{
    public static final String TEST_VALUE = "Test";

    private DisplayFragment fragment;
    private EditText display;

    private Bus bus;

    @Before
    public void setUp() throws Exception
    {
        fragment = DisplayFragment.newInstance();
        startFragment( fragment );

        display = (EditText) fragment.getView()
                    .findViewById( R.id.calculator_display );

        bus = CalculatorApplication.getInstance().getBus();
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( fragment );
    }
}
```

```

@Test
public void shouldHaveDisplay() throws Exception
{
    assertViewIsVisible( display );
}

@Test
public void shouldHaveDefaultDisplay() throws Exception
{
    assertDefaultDisplay();
}

@Test
public void shouldUpdateBlankDisplayAfterAppendEvent() throws Exception
{
    postAppendEvent( TEST_VALUE );
    assertValueDisplayed( TEST_VALUE );
}

@Test
public void appendEventShouldAppendDisplay() throws Exception
{
    setDisplayValue( TEST_VALUE );
    postAppendEvent( TEST_VALUE );
    assertValueDisplayed( TEST_VALUE + TEST_VALUE );
}

private void setDisplayValue( String value )
{
    display.setText( value );
}

@Test
public void shouldSetDisplayOnSetDisplayEvent() throws Exception
{
    postSetDisplayEvent( TEST_VALUE );
    assertValueDisplayed( TEST_VALUE );
}

private void assertValueDisplayed( String value )
{
    assertThat( display.getText().toString(), equalTo( value ) );
}

```

```
private void assertDefaultDisplay()
{
    assertEquals( getString( R.string.DEFAULT_DISPLAY ) );
}

private void postSetDisplayEvent( String value )
{
    bus.post( new SetDisplayEvent( value ) );
}

private void postAppendEvent( String value )
{
    bus.post( new AppendEvent( value ) );
}
```

CalculatorActivity.java

```
package com.greenlifesoftware.calculator;

import android.support.v4.app.FragmentTransaction;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;

import com.greenlifesoftware.calculator.events.DisplayEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.squareup.otto.Bus;
import com.squareup.otto.Subscribe;

public class CalculatorActivity extends ActionBarActivity
{
    public static final String STATE_FRAGMENT_TAG = "CalculatorState";

    @Override
    protected void onCreate( Bundle savedInstanceState )
    {
        super.onCreate( savedInstanceState );
        setContentView( R.layout.activity_calculator );
        addStateFragment();
    }

    private void addStateFragment()
    {
        FragmentTransaction transaction = getSupportFragmentManager()
            .beginTransaction();
        transaction.add( CalculatorStateFragment.newInstance(),
            STATE_FRAGMENT_TAG );
        transaction.commit();
    }
}
```

CalculatorActivityTest.java

```
package com.greenlifesoftware.calculator;

import android.support.v4.app.Fragment;

import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.robolectric.Robolectric;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

@RunWith(RobolectricGradleTestRunner.class)

public class CalculatorActivityTest
{
    private CalculatorActivity activity;

    @Before
    public void setUp() throws Exception
    {
        activity = Robolectric.buildActivity( CalculatorActivity.class )
            .create()
            .start()
            .resume()
            .get();
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( activity );
    }

    @Test
    public void shouldHaveDisplayFragment() throws Exception
    {
        assertNotNull( getFragmentById( R.id.display_fragment ) );
    }
}
```

```
@Test
public void shouldHaveButtonsFragment() throws Exception
{
    assertNotNull( getFragmentById( R.id.buttons_fragment ) );
}

@Test
public void shouldHaveCalculatorStateFragment() throws Exception
{
    Fragment calculatorState = getStateFragment();
    assertNotNull( calculatorState );
    assertTrue( calculatorState instanceof CalculatorStateFragment );
}

private Fragment getStateFragment()
{
    return activity.getSupportFragmentManager()
        .findFragmentByTag( CalculatorActivity.STATE_FRAGMENT_TAG );
}

Fragment getFragmentById( int id )
{
    return activity.getSupportFragmentManager().findFragmentById( id );
}
```

Events

AppendEvent.java

```
package com.greenlifesoftware.calculator.events;

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

SetDisplayEvent.java

```
package com.greenlifesoftware.calculator;

import com.greenlifesoftware.calculator.events.DisplayEvent;

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

Chapter 11

Clearing the Calculator

When a user makes a mistake, such as entering a wrong number or operator, there's currently no way to undo that action. Let's improve the user experience.

In this chapter, we'll clear the currently displayed text and reset the calculator's state to default values when the clear button is pressed.¹

When you press the clear button:

1. `ButtonsFragment` posts a `ClearEvent`.
2. `CalculatorActivity`'s bus broadcasts the event.
3. `DisplayFragment` has registered with the bus and subscribed for the `ClearEvent`. It clears the currently shown text in the display when this event is fired.
4. `CalculatorStateFragment` has registered with the bus and subscribed for the `ClearEvent`. It clears the stored calculator state.

¹We'll follow the paradigm in [Chapter 8](#) for routing and responding to display events.

11.1 Clearing the Display

Add a test named `shouldClearDisplayOnClearEvent()` to `DisplayFragmentTest.java`.

First, set the display text so we can be sure the a clear event restores the display to the default text. Then post a `ClearEvent` to the bus and verify that the displayed text matches the default display string.

```
@Test
public void shouldClearDisplayOnClearEvent() throws Exception
{
    setDisplayValue( TEST_VALUE );
    postClearEvent();
    assertDefaultDisplay();
}
```

Use the  shortcut to create the `postClearEvent()` function. Inside the function, use the bus to post a `ClearEvent`.

```
private void postClearEvent()
{
    bus.post( new ClearEvent() );
}
```

Use  again to create the `ClearEvent` event in the `events` package of the `app` module.

In `ClearEvent.java`, extend `BaseEvent`. We only need the default constructor since we're solely using this event to signal a cleared state.

```
package com.greenlifesoftware.calculator.events;

public class ClearEvent extends BaseEvent
{}
```

The test fails. Let's make it pass.

Post Clear Events

Button clicks are the source of events, so we'll add a test for posting clear events to `ButtonsFragmentTest.java`.

Name the test `clearButtonShouldPostClearEvent()`.

Promote `clearButton` to a field (from the test `shouldHaveClearButton()`) using the shortcut  +  . Inside the test, click the clear button and ensure a clear event is posted.

```
@Test
public void clearButtonShouldPostClearEvent() throws Exception
{
    clearButton.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof ClearEvent );
}
```

In `ButtonsFragment.java`, add a call to `configureClearButton()` from `onCreate()`. Use  +  to generate the function.

In the function, get a reference to the clear button's view and add a click listener. Inside the click listener, simply post a `ClearEvent` to the bus.

```
private void configureClearButton()
{
    layout.findViewById( R.id.button_clear )
        .setOnClickListener( new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            postToBus( new ClearEvent() );
        }
    } );
}
```

Run the tests. Now tests in `ButtonsFragmentTest.java` should pass, but our test in `DisplayFragmentTest.java` still fails.

Clear the Display

Now that we have the `ButtonsFragment` producing the clear event, we can subscribe to `ClearEvents` in `DisplayFragment.java`.

```
@Subscribe  
public void onClearDisplay( ClearEvent event )  
{  
    setDisplay( getString( R.string.DEFAULT_DISPLAY ) );  
}
```

Set the text back to the default text when this event occurs.

Note: We're using `Fragment`'s `getString()` API.

Update the Default Display

For the sake of seeing something in the display, we set the default string to “Display” back in `Test Default Display`.

This isn’t what we want to see when we update the calculator, so update the string `DEFAULT_DISPLAY` in `strings.xml` to an empty string.

```
<string name="DEFAULT_DISPLAY"/>
```

Run the tests, they should pass.

Build the app to verify that the display clears when the the clear button is pressed.

11.2 Clearing the Operand

Although visually it appears that the calculator has been cleared, we still have calculator state to manage.

Add a test to `CalculatorStateFragmentTest.java` named `clearShouldClearOperand()`.

In the test, post a number event followed by a clear event to simulate clearing the displayed value.

```
@Test
public void clearShouldClearOperand() throws Exception
{
    postNumberEvent();
    postClearEvent();
    assertThat( fragment.getOperand(),
                equalTo( "" ) );
}
```

Then ensure that the operand is reset to an empty string.

First, use to generate `postClearEvent()`. Post an event to the bus.

```
private void postClearEvent()
{
    bus.post( new ClearEvent() );
}
```

Run the test.

```
java.lang.AssertionError: Expected: "" but: was "1"
```

Let's add the logic to make the test pass.

Clear the Operand

The fix is simple.

In `CalculatorStateFragment.java`, add a `ClearEvent` subscription and reset `operand` to an empty string.

```
@Subscribe  
public void onClearEvent( ClearEvent event )  
{  
    operand = "";  
}
```

Run the tests again. They should pass.

11.3 Clearing the Operator

Add a test to `CalculatorStateFragmentTest.java` named `clearShouldClearOperator()`.

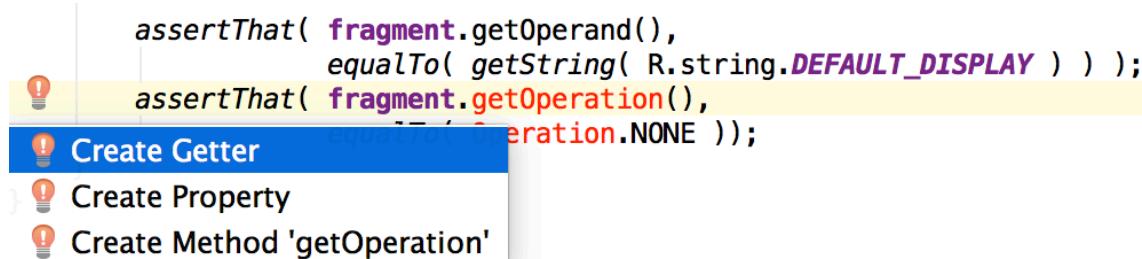
In the test, post a number event followed by a clear event to simulate clearing the displayed value. Then ensure that the operator is the type we expect.

```
@Test  
public void clearShouldClearOperator() throws Exception  
{  
    postNumberEvent();  
    postClearEvent();  
    assertThat( fragment.getOperator(),  
                equalTo( Operator.NONE ));  
}
```

Get Operator

From the test file, use the key combination $\text{Alt} + \text{Shift} + \text{G}$ to create the getter in `CalculatorStateFragment.java`.

Note: If you use the key combination and select `Create Getter`, it creates the field automatically.



In the `getOperator()` signature, while the type is still red, set the return type to `Operator`. Hit  to exit the suggested type completions.

```
public Operation getOperation()
{
    return operation;
}
```

Now hit  to accept the unknown type and  again to finalize the getter and generate the field with the proper type. Finally, change the access modifier to `protected` since we won't be using this outside of testing.

```
protected Operator getOperator()
{
    return operator;
}
```

If necessary, create the field. Set the value of `operator` to `Operator.NONE`.

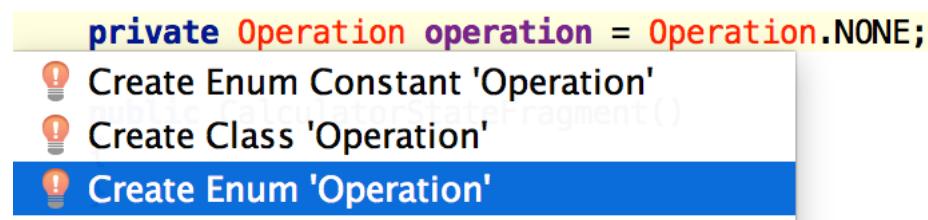
```
private Operator operator = Operator.NONE;
```

Next we'll create  `Operator.java`.

Create Operator Type

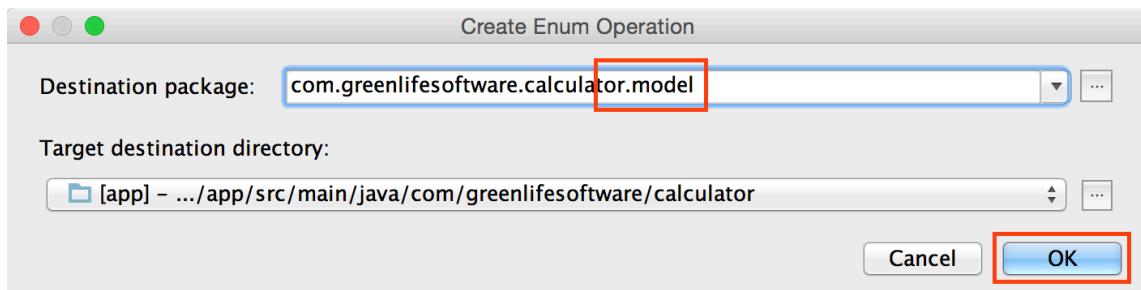
This **Operator** type will help us track the state of our calculator.

From `CalculatorStateFragment.java`, generate the **Operator** type by using the shortcut `Alt + Insert` on the field declaration.



Choose `Create Enum 'Operator'` from the pop-up to open a dialog.

In the dialog's `Destination package` section, add “model” to create a new package.



In `Operator.java`, add the first value, **NONE**.

```
package com.greenlifesoftware.calculator.model;

public enum Operator
{
    NONE
}
```

Import in `Operator` in `CalculatorStateFragmentTest.java`.

Run the tests. They pass because the default value is `Operator.NONE`.

Clear Operator Type

In `CalculatorStateFragment.java`'s clear event subscription, add a line that resets the `operator` type when we reset `operand`'s value.

```
@Subscribe  
public void onClearSelected( ClearEvent event )  
{  
    operand = "";  
    operator = Operator.NONE;  
}
```

Run the tests again. They should still pass.

Run the calculator to make sure the view behaves as expected.

Finally, once everything is working, commit.

Summary

In this chapter, we cleared the display and calculator state when the clear button was pressed. We also created the `Operator` type that we'll use in future chapters.

calculatorstatefragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import com.greenlifesoftware.calculator.events.AppendEvent;
import com.greenlifesoftware.calculator.events.ClearEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;
import com.greenlifesoftware.calculator.events.SetDisplayEvent;

import com.greenlifesoftware.calculator.model.Operator;
import com.squareup.otto.Subscribe;

public class CalculatorStateFragment extends BaseFragment
{
    public static final int MAX_OPERAND_LENGTH = 10;

    private String operand = "";
    private boolean operatorWasPressed = false;
    private Operator operator = Operator.NONE;

    public CalculatorStateFragment()
    {
    }

    public static CalculatorStateFragment newInstance()
    {
        return new CalculatorStateFragment();
    }

    @Nullable
    @Override
    public View onCreateView( LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState )
    {
        return null;
    }
}
```

```

}

@Subscribe
public void onNumberSelected( NumberEvent event )
{
    if (maxOperatorLimitReached())
    {
        return;
    }

    if (operatorWasPressed)
    {
        postToBus( new SetDisplayEvent( event.getNumber() ) );
    }
    else
    {
        postToBus( new AppendEvent( event.getNumber() ) );
    }

    operand += event.getNumber();
    operatorWasPressed = false;
}

private boolean maxOperatorLimitReached()
{
    return operand.length() >= MAX_OPERAND_LENGTH;
}

@Subscribe
public void onOperatorSelected( OperatorEvent event )
{
    operatorWasPressed = true;
    operand = "";
}

@Subscribe
public void onClearSelected( ClearEvent event )
{
    operand = "";
}

protected String getOperand()
{
    return operand;
}

protected Operator getOperator()
{
    return operator;
}
}

```

CalculatorStateFragmentTest.java

```
package com.greenlifesoftware.calculator;

import com.greenlifesoftware.calculator.events.*;
import com.greenlifesoftware.calculator.model.Operator;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;
import com.squareup.otto.Bus;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static com.greenlifesoftware.calculator.CalculatorStateFragment.*;
import static junit.framework.Assert.assertTrue;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.*;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)
public class CalculatorStateFragmentTest
{
    private static final String NUMBER_VALUE = "1";
    private static final String OPERATOR = "%";
    private static final int OPERAND_LENGTH = 3;

    private CalculatorStateFragment fragment;

    private Bus bus;
    private BusHelper busHelper;

    @Before
    public void setUp() throws Exception
    {
        fragment = newInstance();
        startFragment( fragment );

        bus = CalculatorApplication.getInstance().getBus();
        busHelper = new BusHelper();
        bus.register( busHelper );
    }

    @Test
    public void shouldNotBeNull() throws Exception
    {
        assertNotNull( fragment );
    }
}
```

```

@Test
public void numberEventShouldFireAppendEvent() throws Exception
{
    postNumberEvent();
    assertAppendEventWasLast();
}

private void assertAppendEventWasLast()
{
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof AppendEvent );
    assertThat( ((AppendEvent) event).getValue(),
                equalTo( NUMBER_VALUE ) );
}

@Test
public void shouldSetDisplayAfterOperatorEvent() throws Exception
{
    postNumberEvent();
    postOperatorEvent();
    postNumberEvent();
    assertSetDisplayEventWasLast();
}

private void assertSetDisplayEventWasLast()
{
    BaseEvent lastEvent = busHelper.getLastEvent();
    assertTrue( lastEvent instanceof SetDisplayEvent );
    assertThat( ((SetDisplayEvent) lastEvent).getValue(),
                equalTo( NUMBER_VALUE ) );
}

@Test
public void shouldConstructOperand() throws Exception
{
    String expectedOperand = constructDefaultOperand();
    assertOperandEquals( expectedOperand );
}

private String constructDefaultOperand()
{
    return constructOperand( OPERAND_LENGTH );
}

private String constructOperand( int operandLength )
{
    return constructOperandMaybePost( operandLength, true );
}

private void assertOperandEquals( String expectedOperand )
{
    assertThat( fragment.getOperand(),
                equalTo( expectedOperand ) );
}

```

```

@Test
public void operandShouldNotExceedMaxLength() throws Exception
{
    constructTooLongOperand();
    assertThat( fragment.getOperand().length() ,
                equalTo( MAX_OPERAND_LENGTH ) );
}

@Test
public void shouldStoreCorrectMaxOperandValue() throws Exception
{
    constructTooLongOperand();
    assertOperandEquals( maxTestOperand() );
}

private void constructTooLongOperand()
{
    constructOperand( MAX_OPERAND_LENGTH + 2 );
}

private String maxTestOperand()
{
    return constructOperandMaybePost( MAX_OPERAND_LENGTH, true );
}

@Test
public void shouldResetOperandAfterOperator() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent();
    assertOperandEquals( "" );
}

@Test
public void shouldConstructNumberAfterOperator() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent();

    String expectedOperand = constructOperand( OPERAND_LENGTH + 2 );
    assertOperandEquals( expectedOperand );
}

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

private void postOperatorEvent()
{
    bus.post( new OperatorEvent( OPERATOR ) );
}

```

```

private String constructOperandMaybePost( int operandLength, boolean shouldPost )
{
    String operand = "";

    for (int operandCounter = 1;
        operandCounter <= operandLength;
        operandCounter++)
    {
        if (shouldPost)
        {
            postNumberEvent();
        }

        operand += NUMBER_VALUE;
    }

    return operand;
}

@Test
public void clearShouldClearOperand() throws Exception
{
    postNumberEvent();
    postClearEvent();

    assertThat( fragment.getOperand(),
                equalTo( "" ) );
}

@Test
public void clearShouldClearOperator() throws Exception
{
    postNumberEvent();
    postClearEvent();
    assertThat( fragment.getOperator(),
                equalTo( Operator.NONE ) );
}

private void postClearEvent()
{
    bus.post( new ClearEvent() );
}
}

```

ButtonsFragment.java

```
package com.greenlifesoftware.calculator;

// Other imports

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.greenlifesoftware.calculator.events.ClearEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;

public class ButtonsFragment extends Fragment
{
    // ...
    @Nullable @Override
    public View onCreateView( LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState )
    {
        layout = inflater.inflate( R.layout.fragment_buttons, container, false );

        configureNumberButtons();
        configureOperatorButtons();
        configureClearButton();
        configureEqualsButton();

        return layout;
    }

    // ...

    private void configureClearButton()
    {
        layout.findViewById( R.id.button_clear )
            .setOnClickListener( new View.OnClickListener()
        {
            @Override
            public void onClick( View v )
            {
                postToBus( new ClearEvent() );
            }
        } );
    }
}
```

ButtonsFragmentTest.java

```
package com.greenlifesoftware.calculator;

// ...

@RunWith(RobolectricGradleTestRunner.class)

public class ButtonsFragmentTest
{
    // ...
    private Button closeButton;

    @Before
    public void setUp() throws Exception
    {
        // ...
        closeButton = getButtonById( R.id.button_clear );
    }

    // ...

    @Test
    public void shouldHaveClearButton() throws Exception
    {
        assertViewIsVisible( closeButton );
        assertThat( closeButton.getText().toString(),
                    equalTo( getString( R.string.BUTTON_CLEAR ) ) );
    }

    @Test
    public void closeButtonShouldPostClearEvent() throws Exception
    {
        closeButton.performClick();
        assertTrue( busHelper.getLastEvent() instanceof ClearEvent );
    }
}
```

DisplayFragment.java

```
package com.greenlifesoftware.calculator;

// ...
public class DisplayFragment extends Fragment
{
    // ...
    @Subscribe
    public void onClearDisplay( ClearEvent event )
    {
        setDisplay( getString( R.string.DEFAULT_DISPLAY ) );
    }
}
```

DisplayFragmentTest.java

```
package com.greenlifesoftware.calculator;

// ...

@RunWith(RobolectricGradleTestRunner.class)
public class DisplayFragmentTest
{
    // ...
    @Test
    public void shouldClearDisplayOnClearEvent() throws Exception
    {
        display.setText( TEST_VALUE );
        bus.post( new ClearEvent() );
        assertThat( display.getText().toString(),
                    equalTo( getString( R.string.DEFAULT_DISPLAY ) ) );
    }
}
```

📁 ClearEvent.java

```
package com.greenlifesoftware.calculator.events;

public class ClearEvent extends BaseEvent
{}
```

📁 Operator.java

```
package com.greenlifesoftware.calculator.model;

public enum Operator
{
    NONE
}
```

📁 strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--
        <string name="DEFAULT_DISPLAY"/>
    </resources>
```

Chapter 12

Handling Operators

In this chapter we handle the operators and thier state.

We will:

- Update the button configuration for each operator.
- Allow users to change the operator they are using.
- Prevent users from entering an operator before a number.

After we complete this chapter, we will be ready to compute the results in [Chapter 13](#).

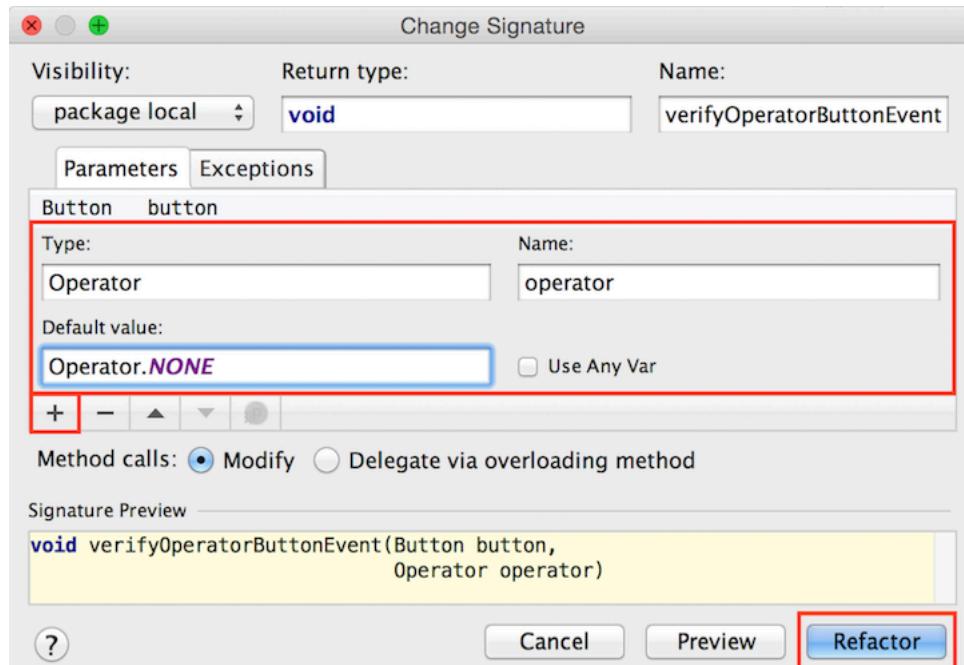
12.1 Using Operator Types

In `ButtonsFragmentTest.java` the operator tests verify that the string configured on the button is sent in the `OperatorEvent`.

```
void verifyOperatorButtonEvent( Button button )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof OperatorEvent );
    assertThat( ((OperatorEvent) event).getOperator(),
               equalTo( button.getText() ) );
}
```

That's not a terribly useful construct when we want to start operating in different ways, depending on which button is pressed. Instead, we will use an `enum` that we can utilize in a `switch` when computing results.

Let's change the test signature by using `⌘ + F6`.



Click the **+** button and enter **Operator** for **Type**, **operator** for **Name**, and **Operator.NONE** for **Default value**. Click **Refactor** to accept.

Note: You could set anything for the default value or nothing at all. We use a **Operator** enum value to make it easier to update the calls for each operator.

Now use **operator.getName()** in the assertion (instead of the button's text). This fetches the string representation of an **enum** value.

```
void verifyOperatorButtonEvent( Button button, Operator operator )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof OperatorEvent );
    assertThat( ((OperatorEvent) event).getOperator(),
               equalTo( operator.getName() ) );
}
```

Run the tests. You get an error like the below for each operator test.

```
java.lang.AssertionError:
Expected: "NONE"
but: was "+"
```

This happens because we are sending the text with each button click instead of the operator's type. Another problem is that we only have **Operator.NONE**.

Let's start with the first problem.

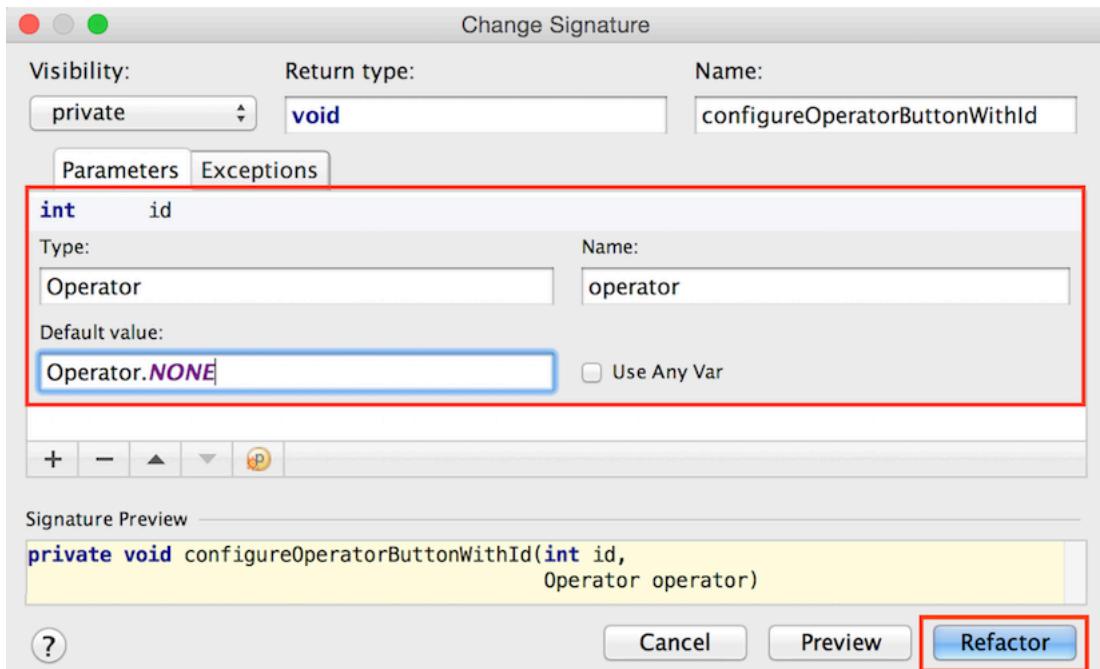
12.2 Handling Operator Events

To send the operator type with the `OperatorEvent`, we need to update the button configuration in `ButtonsFragment.java`.

In the first call to `configureOperatorButtonWithId()`, add `Operator.NONE` as if was already an accepted argument to the function. This creates a build error that we can resolve by using + .

```
private void configureOperatorButtons()
{
    configureOperatorButtonWithId( R.id.button_plus, Operator.NONE );
}
! Add 'Operator' as 2nd parameter to method 'configureOperatorButtonWithId'
```

Select `Add 'Operator' as 2nd parameter to method 'configureOperatorButtonWithId'` from the pop-up.¹ When the dialog opens, change the parameter's name from `none` to `operator` and click `Refactor` to accept the change.



¹When we're done, all the other calls to will use `Operator.NONE`.

Pass the Operator Along

Inside `configureOperatorButtonWithId()`, pass `operator` on to the `createOperatorOnClickListener()` function using the same process. First add the operator to the the call and then use + to change the signature.

Note: We don't have to go through the signature dialog this time because there are no default values involved that would affect other calls to this function.

Update the Click Listener

In `createOperatorOnClickListener()`, we'll send the operator's type when producing the `OperatorEvent`.

You'll notice that we already have a local variable called `operator` that is fetching the button's text. We don't need this anymore, so simply delete it.

```
String operator = ((Button) v).getText().toString();
```

Now we have another issue. If you hover over the red squiggly line, you'll see that the function wants the variable to be final.

```
private View.OnClickListener createOperatorOnClickListener( Operator operator )
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            postToBus( new OperatorEvent( operator ) );
        }
    };
}
```

Variable 'operator' is accessed from within inner class, needs to be declared final

This is a simple fix, just use +.

This introduces another issue – the type that we’re giving to the event is not what it expects. This happens because **OperatorEvent** is currently stored as a **String** instead of an **Operator**.

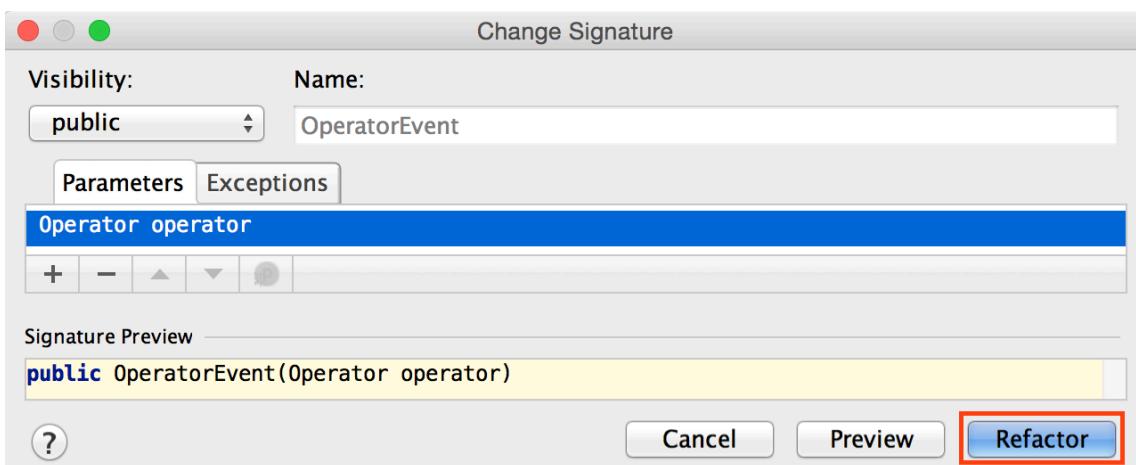
```
private View.OnClickListener createOperatorOnClickListener( final Operator operator )
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            postToBus( new OperatorEvent( operator ) );
        }
    };
}
```

OperatorEvent (java.lang.String)
to (com.greennifeforsoftwae.calculator.model.Operator)

in OperatorEvent cannot be applied

Use the shortcut  +  to change the type.

Select **Change 1st parameter of method 'OperatorEvent' from 'String' to 'Operator'** from the pop-up to bring up the change signature dialog.

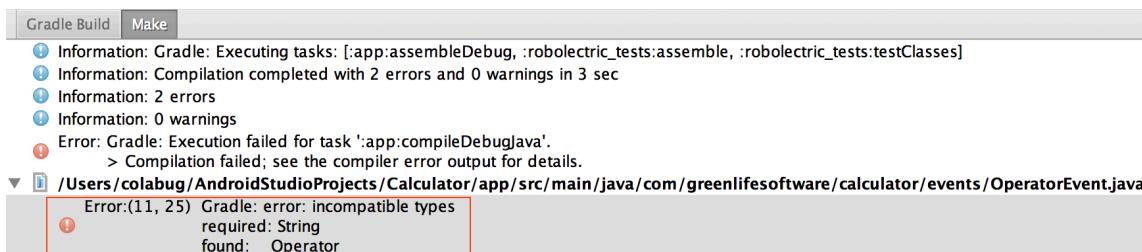


Simply click **Refactor** here to accept the change.

This file is now compiling. Let’s see if there are other things that need to be fixed.

Run the Tests

If you run the tests, you'll find a build error about incompatible types.



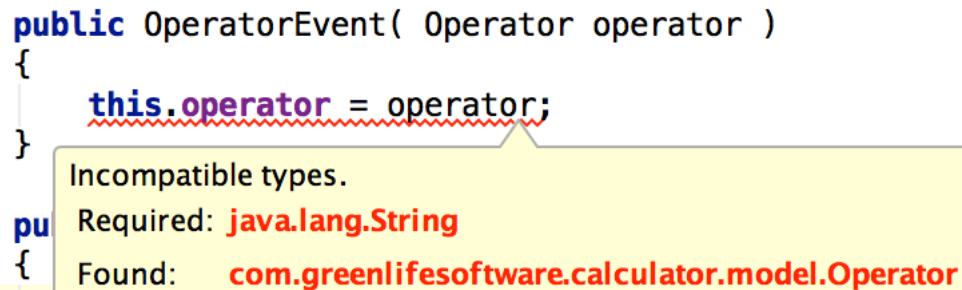
The screenshot shows the Android Studio build output window. It displays the following text:

```
Gradle Build Make
Information: Gradle: Executing tasks: [:app:assembleDebug, :robolectric_tests:assemble, :robolectric_tests:testClasses]
Information: Compilation completed with 2 errors and 0 warnings in 3 sec
Information: 2 errors
Information: 0 warnings
Error: Gradle: Execution failed for task ':app:compileDebugJava'.
> Compilation failed; see the compiler error output for details.
▼ /Users/colabug/AndroidStudioProjects/Calculator/app/src/main/java/com/greenlifesoftware/calculator/events/OperatorEvent.java
Error:(11, 25) Gradle: error: incompatible types
    required: String
        found:    Operator
```

It's complaining about the `OperatorEvent.java` file. Open the file by clicking on the build error or by using $\text{⌘} + \text{↑} + \text{O}$ and typing the file name.

Update the Operator Event

Hover over the red squiggly line to discover what's wrong.



```
public OperatorEvent( Operator operator )
{
    this.operator = operator;
}
Incompatible types.
Required: java.lang.String
Found:    com.greenlifesoftware.calculator.model.Operator
```

We are storing the operator as a `String` instead of an `Operator`, but the constructor is taking an `Operator` type.

Use + to update the field's type.

```
public OperatorEvent( Operator operator )
{
    this.operator = operator;
}
! Change field 'operator' type to 'com.greenlifesoftware.calculator.model.Operator'
! Change parameter 'operator' type to 'java.lang.String'
```

Select .

Once you do that, you'll see yet another error In `getOperator()`.

```
public String getOperator()
{
    return operator;
}
```

Incompatible types.
Required: `java.lang.String`
Found: `com.greenlifesoftware.calculator.model.Operator`

Use + again to return the correct type.

Getting it Building Again

Now that we've swapped the `String` type for an `Operator` type, there's a few more places that are unhappy. Continue to run the tests to work through the build errors.

In `CalculatorStateFragment.java`, we're sending the operator through to the `SetDisplayEvent` in the `onOperatorSelected()` subscription. Change this call to send the name of the operator.

```
@Subscribe  
public void onOperatorSelected( OperatorEvent event )  
{  
    operatorWasPressed = true;  
    operand = "";  
    postToBus( new SetDisplayEvent( event.getOperator().name() ) );  
}
```

In `CalculatorStateFragmentTest.java`, the field `OPERATOR` is set up as a `String`. Change it to an `Operator` type and use `Operator.NONE`.

```
private static final Operator OPERATOR = Operator.NONE;
```

The test `operatorEventShouldFireSetEvent()` wants a string in the call to `assertSetDisplayEventWithValue()`, so use `name()` here as well.

```
@Test  
public void operatorEventShouldFireSetEvent() throws Exception  
{  
    postOperatorEvent();  
    assertSetDisplayEventWithValue( OPERATOR.name() );  
}
```

Finally, in `ButtonsFragmentTest.java`, we're using a string, but really we'd like to compare operator types in `verifyOperatorButtonEvent()`. Remove the call to `name()`.

```
void verifyOperatorButtonEvent( Button button, Operator operator )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof OperatorEvent );
    assertThat( ((OperatorEvent) event).getOperator(),
               equalTo( operator ) );
}
```

Now all tests should be building and passing.

The only problem is that this is a lie – each operator button is sending the wrong type: `Operator.NONE`.

Now we'll create new operator types and send them when an operator is clicked.

12.3 Adding Operator Types

Now it's time to extend `Operator.java` that we created in [Chapter 11](#) to add values for each of the operator keys.

```
public enum Operator
{
    PLUS, MINUS,
    MULTIPLY, DIVIDE, MODULO,
    NONE
}
```

Now update all the operator tests in `ButtonsFragmentTest.java` to test for the right type. For example, `plusButtonShouldPostEvent()` should use `Operator.PLUS` in its call to `verifyOperatorButtonEvent()`.

```
@Test
public void plusButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( plus, Operator.PLUS );
}
```

Running the tests will show you which ones still need to be updated.

```
java.lang.AssertionError:
Expected: <PLUS>
      but: was <NONE>
```

To make a test pass, update `configureOperatorButtons()` with the right type in `ButtonsFragment.java`.

```
configureOperatorButtonWithId( R.id.button_plus, Operator.PLUS );
```

When you've updated all the types, all the tests should pass.

Add Operator Strings

Now run the application. It's not doing what we expect. When we click a key, it updates the view with the name of the enum (e.g. **MULTIPLY** instead of *****).

With enums, you can create constructors and other methods.² We'll use this capability to show our enum properly in the calculator's display.

Create a constructor that takes a **String** named **operatorString**, a field to store the value, and a getter to retrieve it called **getOperatorString()**.

```
public enum Operator
{
    PLUS( "+" ), MINUS( "-" ),
    MULTIPLY( "*" ), DIVIDE( "/" ), MODULO( "%" ),
    NONE( "" );

    private String operatorString;

    Operator( String operatorString )
    {
        this.operatorString = operatorString;
    }

    public String getOperatorString()
    {
        return operatorString;
    }
}
```

For each operator, use the constructor and add the string that will be used in the calculator's display.

²Check out [Oracle's documentation](#) for more information.

Update the Test

In `CalculatorStateFragment.java` we are sending the string to be displayed, so let's update the tests.

First, update the field `OPERATOR` to use a type other than `NONE`.

```
private static final Operator OPERATOR = Operator.MODULO;
```

In the `operatorEventShouldFireSetEvent()` test, update the call to `name()` to `getOperatorString()`.

```
@Test
public void operatorEventShouldFireSetEvent() throws Exception
{
    postOperatorEvent();
    assertSetDisplayEventWithValue( OPERATOR.getOperatorString() );
}
```

Running the test shows that `CalculatorStateFragment.java` needs an update.

```
java.lang.AssertionError:
Expected: "%"
but: was "MODULO"
```

To fix, update `onOperatorSelected()` to use the operator string.

```
@Subscribe
public void onOperatorSelected( OperatorEvent event )
{
    operatorWasPressed = true;
    operand = "";
    postToBus( new SetDisplayEvent( event.getOperator().getOperatorString() ) );
}
```

When you run the tests, they should pass. The app behaves as expected.

12.4 Changing Operators

Once you've entered a number, you should be able to change your mind on which operator you'd like to use until you enter a second operand.

Currently, when we click on new operators, the display behaves as expected. Let's make sure that's true behind the scenes as well.

Add a test to `CalculatorStateFragmentTest.java` called `shouldBeAbleToChangeOperators()`.

In this test, construct an operand³, post an operator event, and then post a different operator event. Assert that the second operator we sent is being saved.

```
@Test
public void shouldBeAbleToChangeOperators() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent( OPERATOR );
    postOperatorEvent( OPERATOR2 );
    assertThat( fragment.getOperator(), equalTo( OPERATOR2 ) );
}
```

You'll notice that we extended `postOperatorEvent()` to take an operator value. If you add `OPERATOR` to the method call and use + , you can add an operator type to the method signature. Another option is using + to change the signature. Then use the value passed in to post an operator.

```
private void postOperatorEvent( Operator operator )
{
    bus.post( new OperatorEvent( operator ) );
}
```

³This is a prerequisite for using an operator. We could skip the first call to construct an operand, but if we do, we'll just have to change this test after we tackle the next section.

Finally add the constant field **OPERATOR2** using the + key combination.

```
private static final Operator OPERATOR2 = Operator.PLUS;
```

Set it to something different than the value for **OPERATOR** (e.g. **PLUS**).

Now run the tests. You should see this error:

```
java.lang.AssertionError:  
Expected: <PLUS>  
      but: was <NONE>
```

This is happening because we are not storing the value when an operator is selected. Let's update the state fragment.

Storing Operator State

In  `CalculatorStateFragment.java`, navigate to the subscription `onOperatorSelected()`. You'll notice no mention of **operator** here.

This is an easy fix. Simply store the operator's value from the event.

```
@Subscribe  
public void onOperatorSelected( OperatorEvent event )  
{  
    operatorWasPressed = true;  
    operand = "";  
    operator = event.getOperator();  
    postToBus( new SetDisplayEvent( event.getOperator().getOperatorString() ) );  
}
```

Re rerun the tests, they should pass.

12.5 Operator First

Right now there's nothing preventing a user from starting the calculator and immediately pressing an operator (e.g. plus). That would put the calculator in a weird state, so let's ignore any operators entered before a number.

Note: We're ignoring the valid entry of minus for simplicity.

Add a test to `CalculatorStateFragmentTest.java` named `shouldNotUpdateOperatorOrOperandBeforeANumber()`. In the test, post an operator and verify that the values for the operator and operand do not change from their default values.

```
@Test
public void shouldNotUpdateOperatorOrOperandBeforeANumber() throws Exception {
    postOperatorEvent( OPERATOR );
    assertThat( fragment.getOperator(), equalTo( Operator.NONE ) );
    assertThat( fragment.getOperand(), equalTo( "" ) );
}
```

When you run the tests, you should see this error:

```
java.lang.AssertionError:
Expected: <NONE>
      but: was <MODULO>
```

This happens because we are unconditionally processing operators.

Let's fix that.

No Operators Before Numbers

In `CalculatorStateFragment.java`, this state is signaled by default values for the operand and operator.

Let's add new logic to `onOperatorSelected()` to check for this state.

```
@Subscribe  
public void onOperatorSelected( OperatorEvent event )  
{  
    if( noNumberEntered() )  
    {  
        return;  
    }  
  
    operatorWasPressed = true;  
    operand = "";  
    operator = event.getOperator();  
    postToBus( new SetDisplayEvent( event.getOperator().getOperatorString() ) );  
}
```

When we're in the default state (no number entered), we don't want to post an event or save the value of the operator. So we simply `return`. To generate the helper, use + . It doesn't take any parameters and returns a `boolean`.

```
private boolean noNumberEntered()  
{  
    return operator == Operator.NONE &&  
        operand.equals( "" );  
}
```

Inside, compare the value of the operator and operand to the defaults.

- The `operator` is set to `NONE` before a number is entered and is reset to `NONE` when the calculator is cleared.
- The `operand` starts with a default value of `""` and is reset on clear.

Run all the tests in `CalculatorStateFragment.java`. Our new test passes, but broke an existing test.

Update Outdated Test

The test `operatorEventShouldFireSetEvent()` is failing because we entered an operator before an operand.

```
junit.framework.AssertionFailedError  
at ...assertSetDisplayEventWithValue(CalculatorStateFragmentTest.java:83)  
at ...operatorEventShouldFireSetEvent(CalculatorStateFragmentTest.java:68)
```

This is no longer a valid state, let's update the test.

Add a call to `constructDefaultOperand()` before posting an operator.

```
@Test  
public void operatorEventShouldFireSetEvent() throws Exception  
{  
    constructDefaultOperand();  
    postOperatorEvent( OPERATOR );  
    assertSetDisplayEventWithValue( OPERATOR.getOperatorString() );  
}
```

Run the tests again. They should pass.

Testing for Set Events

We've made sure that the values are not updated, but we're not making sure that display events weren't sent in this state.

Add another test to verify the display is untouched named

`operatorShouldNotFireSetDisplayEventBeforeANumberIsEntered()`.

```
@Test
public void operatorShouldNotFireSetDisplayEventBeforeANumberIsEntered()
    throws Exception
{
    postOperatorEvent( OPERATOR );
    assertThat( busHelper.getEventListSize(), equalTo( 1 ) );
    assertTrue( busHelper.getLastEvent() instanceof OperatorEvent );
}
```

In the test, post an operator, check the size of the `busHelper`'s list of events, and ensure the last one was an `OperatorEvent` – not a `SetDisplayEvent`.

Note: The `OperatorEvent` still happens, but we ignore it.

Use  +  to generate a helper function in  `BusHelper.java` named `getEventListSize()`. Simply return the size of the array as an `int`.

```
public int getEventListSize()
{
    return events.size();
}
```

Run all tests again in our application, they should be passing.

Refactor

Now that we're in the green state, we can refactor to our heart's content.

I noticed duplication in `CalculatorStateFragment.java` when we are setting or referring to the default value for `operand` (which is `""`). This occurs when clearing, setting the default value, and checking the default state.

Instead, let's create a constant field called `DEFAULT_OPERAND` and use it where we were previously using `""`.⁴

```
public static final String DEFAULT_OPERAND = "";
private String operand = DEFAULT_OPERAND;

//...

private boolean noNumberEntered()
{
    return operator == Operator.NONE &&
        operand.equals( DEFAULT_OPERAND );
}

@Subscribe
public void onClearSelected( ClearEvent event )
{
    operand = DEFAULT_OPERAND;
    operator = Operator.NONE;
}
```

Note: If you make the field public, you can also use it in the tests. Simply search for the string `""` and replace it with `DEFAULT_OPERAND` where it makes sense.

Run the tests. They should still pass.

⁴This has the added value of increasing readability.

Summary

In this chapter we managed the operators of our calculator. We:

- Updated the button configuration to use operator types (Sections [Using Operator Types](#) and [Handling Operator Events](#)).
- Created additional operator types (Section [Adding Operator Types](#)).
- Allowed the user to change their mind when selecting operators in the middle of the calculation (Section [Changing Operators](#)).
- Guarded against an improper state of entering an operator before a number (Section [Operator First](#)).

Now we have all the ingredients in place to calculate results!

CalculatorStateFragmentTest.java

```
package com.greenlifesoftware.calculator;

import com.greenlifesoftware.calculator.events.*;
import com.greenlifesoftware.calculator.model.Operator;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;
import com.squareup.otto.Bus;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static com.greenlifesoftware.calculator.CalculatorStateFragment.*;
import static junit.framework.Assert.assertTrue;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.*;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)
public class CalculatorStateFragmentTest
{
    private static final String NUMBER_VALUE = "1";

    private static final Operator OPERATOR = Operator.MODULO;
    private static final Operator OPERATOR2 = Operator.PLUS;

    private static final int OPERAND_LENGTH = 3;

    private CalculatorStateFragment fragment;

    private Bus bus;
    private BusHelper busHelper;

    @Before
    public void setUp() throws Exception
    {
        fragment = newInstance();
        startFragment( fragment );

        bus = CalculatorApplication.getInstance().getBus();
        busHelper = new BusHelper();
        bus.register( busHelper );
    }
}
```

```

@Test
public void shouldNotBeNull() throws Exception
{
    assertNotNull( fragment );
}

@Test
public void numberEventShouldFireAppendEvent() throws Exception
{
    postNumberEvent();
    assertAppendEventWasLast();
}

private void assertAppendEventWasLast()
{
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof AppendEvent );
    assertThat( ((AppendEvent) event).getValue(),
               equalTo( NUMBER_VALUE ) );
}

@Test
public void operatorEventShouldFireSetEvent() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent( OPERATOR );
    assertSetDisplayEventWithValue( OPERATOR.getOperatorString() );
}

@Test
public void operatorShouldNotFireSetDisplayEventBeforeANumberIsEntered()
    throws Exception
{
    postOperatorEvent( OPERATOR );
    assertThat( busHelper.getEventListSize(), equalTo( 1 ) );
    assertTrue( busHelper.getLastEvent() instanceof OperatorEvent );
}

@Test
public void shouldSetDisplayAfterOperatorEvent() throws Exception
{
    postNumberEvent();
    postOperatorEvent( OPERATOR );
    postNumberEvent();
    assertSetDisplayEventWithValue( NUMBER_VALUE );
}

```

```

private void assertSetDisplayEventWithValue( String value )
{
    BaseEvent lastEvent = busHelper.getLastEvent();
    assertTrue( lastEvent instanceof SetDisplayEvent );
    assertThat( ((SetDisplayEvent) lastEvent).getValue(),
                equalTo( value ) );
}

@Test
public void shouldConstructOperand() throws Exception
{
    String expectedOperand = constructDefaultOperand();
    assertOperandEquals( expectedOperand );
}

private String constructDefaultOperand()
{
    return constructOperand( OPERAND_LENGTH );
}

private String constructOperand( int operandLength )
{
    return constructOperandMaybePost( operandLength, true );
}

private void assertOperandEquals( String expectedOperand )
{
    assertThat( fragment.getOperand(),
                equalTo( expectedOperand ) );
}

@Test
public void operandShouldNotExceedMaxLength() throws Exception
{
    constructTooLongOperand();
    assertThat( fragment.getOperand().length(),
                equalTo( MAX_OPERAND_LENGTH ) );
}

@Test
public void shouldStoreCorrectMaxOperandValue() throws Exception
{
    constructTooLongOperand();
    assertOperandEquals( maxTestOperand() );
}

private void constructTooLongOperand()
{
    constructOperand( MAX_OPERAND_LENGTH + 2 );
}

private String maxTestOperand()
{
    return constructOperandMaybePost( MAX_OPERAND_LENGTH, true );
}

```

```

@Test
public void shouldResetOperandAfterOperator() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent( OPERATOR );
    assertOperandEquals( DEFAULT_OPERAND );
}

@Test
public void shouldConstructNumberAfterOperator() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent( OPERATOR );

    String expectedOperand = constructOperand( OPERAND_LENGTH + 2 );
    assertOperandEquals( expectedOperand );
}

@Test
public void shouldBeAbleToChangeOperators() throws Exception
{
    constructDefaultOperand();
    postOperatorEvent( OPERATOR );
    postOperatorEvent( OPERATOR2 );
    assertThat( fragment.getOperator(), equalTo( OPERATOR2 ) );
}

@Test
public void shouldNotUpdateOperatorOrOperandBeforeANumber() throws Exception
{
    postOperatorEvent( OPERATOR );
    assertThat( fragment.getOperand(), equalTo( DEFAULT_OPERAND ) );
    assertThat( fragment.getOperator(), equalTo( Operator.NONE ) );
}

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

private void postOperatorEvent( Operator operator )
{
    bus.post( new OperatorEvent( operator ) );
}

```

```

private String constructOperandMaybePost( int operandLength,
                                         boolean shouldPost )
{
    String operand = DEFAULT_OPERAND;

    for (int operandCounter = 1;
         operandCounter <= operandLength;
         operandCounter++)
    {
        if (shouldPost)
        {
            postNumberEvent();
        }

        operand += NUMBER_VALUE;
    }

    return operand;
}

@Test
public void clearShouldClearOperand() throws Exception
{
    postNumberEvent();
    postClearEvent();
    assertThat( fragment.getOperand(),
                equalTo( DEFAULT_OPERAND ) );
}

@Test
public void clearShouldClearOperator() throws Exception
{
    postNumberEvent();
    postClearEvent();
    assertThat( fragment.getOperator(),
                equalTo( Operator.NONE ) );
}

private void postClearEvent()
{
    bus.post( new ClearEvent() );
}
}

```

CalculatorStateFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import com.greenlifesoftware.calculator.events.*;
import com.greenlifesoftware.calculator.model.Operator;

import com.squareup.otto.Subscribe;

public class CalculatorStateFragment extends BaseFragment
{
    public static final int MAX_OPERAND_LENGTH = 10;
    public static final String DEFAULT_OPERAND = "";

    private String operand = DEFAULT_OPERAND;
    private boolean operatorWasPressed = false;
    private Operator operator = Operator.NONE;

    public CalculatorStateFragment()
    {
    }

    public static CalculatorStateFragment newInstance()
    {
        return new CalculatorStateFragment();
    }

    @Override @Nullable
    public View onCreateView( LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState )
    {
        return null;
    }
}
```

```

@Subscribe
public void onNumberSelected( NumberEvent event )
{
    if (maxOperatorLimitReached())
    {
        return;
    }

    if (operatorWasPressed)
    {
        postToBus( new SetDisplayEvent( event.getNumber() ) );
    }
    else
    {
        postToBus( new AppendEvent( event.getNumber() ) );
    }

    operand += event.getNumber();
    operatorWasPressed = false;
}

private boolean maxOperatorLimitReached()
{
    return operand.length() >= MAX_OPERAND_LENGTH;
}

@Subscribe
public void onOperatorSelected( OperatorEvent event )
{
    if (noNumberEntered())
    {
        return;
    }

    operatorWasPressed = true;
    operand = DEFAULT_OPERAND;
    operator = event.getOperator();
    postToBus( new SetDisplayEvent( event.getOperator().getOperatorString() ) );
}

private boolean noNumberEntered()
{
    return operator == Operator.NONE &&
           operand.equals( DEFAULT_OPERAND );
}

```

```
@Subscribe
public void onClearSelected( ClearEvent event )
{
    operand = DEFAULT_OPERAND;
    operator = Operator.NONE;
}

protected String getOperand()
{
    return operand;
}

protected Operator getOperator()
{
    return operator;
}
```

ButtonsFragmentTest.java

```
package com.greenlifesoftware.calculator;

import android.view.View;
import android.widget.Button;

import com.greenlifesoftware.calculator.events.*;
import com.greenlifesoftware.calculator.model.Operator;
import com.greenlifesoftware.calculator.support.BusHelper;
import com.greenlifesoftware.calculator.support.RobolectricGradleTestRunner;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.robolectric.shadows.ShadowToast;

import static com.greenlifesoftware.calculator.support.ResourceLocator.getString;
import static org.hamcrest.core.AreEqual.equalTo;
import static org.junit.Assert.assertNotNull;
import static com.greenlifesoftware.calculator.support.Assert.assertViewIsVisible;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;
import static org.robolectric.util.FragmentTestUtil.startFragment;

@RunWith(RobolectricGradleTestRunner.class)

public class ButtonsFragmentTest
{
    private ButtonsFragment fragment;

    private BusHelper busHelper;

    // Numbers
    private Button button1;
    private Button button2;
    private Button button3;
    private Button button4;
    private Button button5;
    private Button button6;
    private Button button7;
    private Button button8;
    private Button button9;
    private Button button0;

    // Operators
    private Button modulo;
    private Button divide;
    private Button minus;
    private Button multiply;
    private Button plus;
```

```

private Button closeButton;
private Button equalsButton;

@Before
public void setUp() throws Exception
{
    fragment = ButtonsFragment.newInstance();
    startFragment( fragment );

    busHelper = new BusHelper();
    CalculatorApplication.getInstance().getBus().register( busHelper );

    // Numbers
    button1 = getButtonById( R.id.button1 );
    button2 = getButtonById( R.id.button2 );
    button3 = getButtonById( R.id.button3 );
    button4 = getButtonById( R.id.button4 );
    button5 = getButtonById( R.id.button5 );
    button6 = getButtonById( R.id.button6 );
    button7 = getButtonById( R.id.button7 );
    button8 = getButtonById( R.id.button8 );
    button9 = getButtonById( R.id.button9 );
    button0 = getButtonById( R.id.button0 );

    // Operators
    plus = getButtonById( R.id.button_plus );
    minus = getButtonById( R.id.button_minus );
    multiply = getButtonById( R.id.button_multiply );
    divide = getButtonById( R.id.button_divide );
    modulo = getButtonById( R.id.button_modulo );

    closeButton = getButtonById( R.id.button_clear );
    equalsButton = getButtonById( R.id.button_equals );
}

//region Presence Verifications
@Test
public void shouldNotBeNull() throws Exception
{
    assertNotNull( fragment );
}

@Test
public void shouldHaveButtons() throws Exception
{
    assertViewIsVisible( getViewById( R.id.calculator_buttons ) );
}

@Test
public void shouldHaveOneButton() throws Exception
{
    assertViewIsVisible( button1 );
    assertThat( button1.getText().toString(),
                equalTo( getString( R.string.BUTTON_1 ) ) );
}

```

```

@Test
public void shouldHaveTwoButton() throws Exception
{
    assertViewIsVisible( button2 );
    assertThat( button2.getText().toString(),
                equalTo( getString( R.string.BUTTON_2 ) ) );
}

@Test
public void shouldHaveThreeButton() throws Exception
{
    assertViewIsVisible( button3 );
    assertThat( button3.getText().toString(),
                equalTo( getString( R.string.BUTTON_3 ) ) );
}

@Test
public void shouldHaveFourButton() throws Exception
{
    assertViewIsVisible( button4 );
    assertThat( button4.getText().toString(),
                equalTo( getString( R.string.BUTTON_4 ) ) );
}

@Test
public void shouldHaveFiveButton() throws Exception
{
    assertViewIsVisible( button5 );
    assertThat( button5.getText().toString(),
                equalTo( getString( R.string.BUTTON_5 ) ) );
}

@Test
public void shouldHaveSixButton() throws Exception
{
    assertViewIsVisible( button6 );
    assertThat( button6.getText().toString(),
                equalTo( getString( R.string.BUTTON_6 ) ) );
}

@Test
public void shouldHaveSevenButton() throws Exception
{
    assertViewIsVisible( button7 );
    assertThat( button7.getText().toString(),
                equalTo( getString( R.string.BUTTON_7 ) ) );
}

@Test
public void shouldHaveEightButton() throws Exception
{
    assertViewIsVisible( button8 );
    assertThat( button8.getText().toString(),
                equalTo( getString( R.string.BUTTON_8 ) ) );
}

```

```

@Test
public void shouldHaveNineButton() throws Exception
{
    assertViewIsVisible( button9 );
    assertThat( button9.getText().toString(),
                equalTo( getString( R.string.BUTTON_9 ) ) );
}

@Test
public void shouldHaveZeroButton() throws Exception
{
    assertViewIsVisible( button0 );
    assertThat( button0.getText().toString(),
                equalTo( getString( R.string.BUTTON_0 ) ) );
}

@Test
public void shouldHavePlusOperator() throws Exception
{
    assertViewIsVisible( plus );
    assertThat( plus.getText().toString(),
                equalTo( getString( R.string.BUTTON_PLUS ) ) );
}

@Test
public void shouldHaveMinusOperator() throws Exception
{
    assertViewIsVisible( minus );
    assertThat( minus.getText().toString(),
                equalTo( getString( R.string.BUTTON_MINUS ) ) );
}

@Test
public void shouldHaveMultiplyOperator() throws Exception
{
    assertViewIsVisible( multiply );
    assertThat( multiply.getText().toString(),
                equalTo( getString( R.string.BUTTON_MULTIPLY ) ) );
}

@Test
public void shouldHaveDivideOperator() throws Exception
{
    assertViewIsVisible( divide );
    assertThat( divide.getText().toString(),
                equalTo( getString( R.string.BUTTON_DIVIDE ) ) );
}

@Test
public void shouldHaveModuloOperator() throws Exception
{
    assertViewIsVisible( modulo );
    assertThat( modulo.getText().toString(),
                equalTo( getString( R.string.BUTTON_MODULO ) ) );
}

```

```

@Test
public void shouldHaveEqualsButton() throws Exception
{
    assertViewIsVisible( equalsButton );
    assertThat( equalsButton.getText().toString(),
                equalTo( getString( R.string.BUTTON_EQUALS ) ) );
}

@Test
public void shouldHaveClearButton() throws Exception
{
    assertViewIsVisible( clearButton );
    assertThat( clearButton.getText().toString(),
                equalTo( getString( R.string.BUTTON_CLEAR ) ) );
}
//endregion

//region Number Event Verifications
@Test
public void oneButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button1 );
}

@Test
public void twoButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button2 );
}

@Test
public void threeButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button3 );
}

@Test
public void fourButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button4 );
}

@Test
public void fiveButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button5 );
}

@Test
public void sixButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button6 );
}

```

```

@Test
public void sevenButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button7 );
}

@Test
public void eightButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button8 );
}

@Test
public void nineButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button9 );
}

@Test
public void zeroButtonShouldPostEvent() throws Exception
{
    verifyNumberEvent( button0 );
}

void verifyNumberEvent( Button button )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof NumberEvent );
    assertThat( ((NumberEvent) event).getNumber(),
               equalTo( button.getText() ) );
}
//endregion

//region Operator Event Verifications
@Test
public void plusButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( plus, Operator.PLUS );
}

@Test
public void minusButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( minus, Operator_MINUS );
}

@Test
public void multiplyButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( multiply, Operator.MULTIPLY );
}

```

```

@Test
public void divideButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( divide, Operator.DIVIDE );
}

@Test
public void moduloButtonShouldPostEvent() throws Exception
{
    verifyOperatorButtonEvent( modulo, Operator.MODULO );
}

void verifyOperatorButtonEvent( Button button, Operator operator )
{
    button.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof OperatorEvent );
    assertThat( ((OperatorEvent) event).getOperator(),
               equalTo( operator ) );
}
//endregion

//region Other Verifications

@Test
public void clearButtonShouldPostClearEvent() throws Exception
{
    clearButton.performClick();
    BaseEvent event = busHelper.getLastEvent();
    assertTrue( event instanceof ClearEvent );
}

@Test
public void equalsButtonShouldToast() throws Exception
{
    equalsButton.performClick();
    assertThat( ShadowToast.getTextOfLatestToast(),
               equalTo( equalsButton.getText() ) );
}
//endregion

//region Helper Functions
private Button getButtonById( int id )
{
    return (Button) getViewById( id );
}

private View getViewById( int id )
{
    return fragment.getView().findViewById( id );
}
//endregion
}

```

ButtonsFragment.java

```
package com.greenlifesoftware.calculator;

import android.os.Bundle;
import android.support.annotation.Nullable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.Toast;

import com.greenlifesoftware.calculator.events.ClearEvent;
import com.greenlifesoftware.calculator.events.NumberEvent;
import com.greenlifesoftware.calculator.events.OperatorEvent;
import com.greenlifesoftware.calculator.model.Operator;

public class ButtonsFragment extends BaseFragment
{
    private View layout;

    public ButtonsFragment()
    {
    }

    @Nullable
    @Override
    public View onCreateView( LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState )
    {
        layout = inflater.inflate( R.layout.fragment_buttons, container, false );

        configureNumberButtons();
        configureOperatorButtons();
        configureClearButton();
        configureEqualsButton();

        return layout;
    }

    private void configureClearButton()
    {
        layout.findViewById( R.id.button_clear )
            .setOnClickListener( new View.OnClickListener()
        {
            @Override
            public void onClick( View v )
            {
                postToBus( new ClearEvent() );
            }
        } );
    }
}
```

```

private void configureNumberButtons()
{
    configureNumberButtonWithId( R.id.button1 );
    configureNumberButtonWithId( R.id.button2 );
    configureNumberButtonWithId( R.id.button3 );
    configureNumberButtonWithId( R.id.button4 );
    configureNumberButtonWithId( R.id.button5 );
    configureNumberButtonWithId( R.id.button6 );
    configureNumberButtonWithId( R.id.button7 );
    configureNumberButtonWithId( R.id.button8 );
    configureNumberButtonWithId( R.id.button9 );
    configureNumberButtonWithId( R.id.button0 );
}

private void configureNumberButtonWithId( int id )
{
    layout.findViewById( id )
        .setOnClickListener( createNumberOnClickListener() );
}

private View.OnClickListener createNumberOnClickListener()
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            String number = ( (Button) v ).getText().toString();
            postToBus( new NumberEvent( number ) );
        }
    };
}

private void configureOperatorButtons()
{
    configureOperatorButtonWithId( R.id.button_plus, Operator.PLUS );
    configureOperatorButtonWithId( R.id.button_minus, Operator_MINUS );
    configureOperatorButtonWithId( R.id.button_multiply, Operator.MULTIPLY );
    configureOperatorButtonWithId( R.id.button_divide, Operator.DIVIDE );
    configureOperatorButtonWithId( R.id.button_modulo, Operator.MODULO );
}

private void configureOperatorButtonWithId( int id, Operator operator )
{
    layout.findViewById( id )
        .setOnClickListener( createOperatorOnClickListener( operator ) );
}

```

```
private View.OnClickListener
    createOperatorOnClickListener( final Operator operator )
{
    return new View.OnClickListener()
    {
        @Override
        public void onClick( View v )
        {
            postToBus( new OperatorEvent( operator ) );
        }
    };
}

private void configureEqualsButton()
{
    layout.findViewById( R.id.button_equals )
        .setOnClickListener( new View.OnClickListener()
        {
            @Override
            public void onClick( View v )
            {
                Toast.makeText( getActivity(),
                    ( (Button) v ).getText(),
                    Toast.LENGTH_SHORT )
                    .show();
            }
        } );
}

public static ButtonsFragment newInstance()
{
    return new ButtonsFragment();
}
}
```

Operator.java

```
package com.greenlifesoftware.calculator.model;

public enum Operator
{
    PLUS( "+" ), MINUS( "-" ),
    MULTIPLY( "*" ), DIVIDE( "/" ), MODULO( "%" ),
    NONE( "" );

    private String operatorString;

    Operator( String operatorString )
    {
        this.operatorString = operatorString;
    }

    public String getOperatorString()
    {
        return operatorString;
    }
}
```

OperatorEvent.java

```
package com.greenlifesoftware.calculator.events;

import com.greenlifesoftware.calculator.model.Operator;

public class OperatorEvent extends BaseEvent
{
    private final Operator operator;

    public OperatorEvent( Operator operator )
    {
        this.operator = operator;
    }

    public Operator getOperator()
    {
        return operator;
    }
}
```

BusHelper.java

```
package com.greenlifesoftware.calculator.support;

import com.greenlifesoftware.calculator.events.BaseEvent;
import com.squareup.otto.Subscribe;

import java.util.ArrayList;

public class BusHelper
{
    private ArrayList<BaseEvent> events = new ArrayList<>();

    public BaseEvent getLastEvent()
    {
        if (!events.isEmpty())
        {
            return events.get( events.size() - 1 );
        }

        return null;
    }

    @Subscribe
    public void onAnyEvent( BaseEvent event )
    {
        events.add( event );
    }

    public int getEventListSize()
    {
        return events.size();
    }
}
```

Chapter 13

Computing Results

Chapter 14

Error States

Chapter 15

Scaling the View

15.1 Flexible Dimensions

In this lab, we'll use the combination of the qualifier system (Section ??) and dimensions to scale our view for different form factors.

Let's create a folder called `values-w350dp` and a `dimens.xml` in that folder. Here is where you'd put your values you'd like to use on screens with at least `350dp` in screen width.

We'll use the [common configuration examples](#) suggested by Android's documentation.

Portrait Sizes

- 320dp: a typical phone screen
- 480dp: a tweener tablet like the [Streak](#)
- 600dp: a 7 tablet
- 720dp: a 10 tablet

480dp Dimensions

```
<resources>
    <!--Buttons-->
    <dimen name="button_dimensions">78dp</dimen>
    <dimen name="button_margin">4dp</dimen>
    <dimen name="button_text_size">50sp</dimen>

    <!--Zero-->
    <dimen name="zero_button_width">164dp</dimen>

    <!--Equals-->
    <dimen name="equals_button_width">250dp</dimen>

    <!--Display-->
    <dimen name="display_margin">8dp</dimen>
    <dimen name="display_text_size">40sp</dimen>
</resources>
```

You don't have to copy all the dimensions over from `values/dimens.xml` - just the ones that you are interested in overwriting.

600dp Dimensions

Adjust Display

For testing, you can use physical devices or an emulator. For best results, use a `small_phone`, a Nexus 4/5, a Nexus 7, and a Nexus 10 to verify the dimensions scale as you expect.

I adjusted my display margins after adjusting all my buttons to make the edges line up better.

Commit.

15.2 Device Rotation

When the device rotates, our view is not pretty. We're also losing or altering the calculator fragment state.

Let's fix it.

Relevant articles:

- [Recreating the activity's state](#)
- [Handling runtime changes](#)
- [Coordinating Activity and Fragment Lifecycles](#)

15.3 Landscape

Chapter 16

Appendices

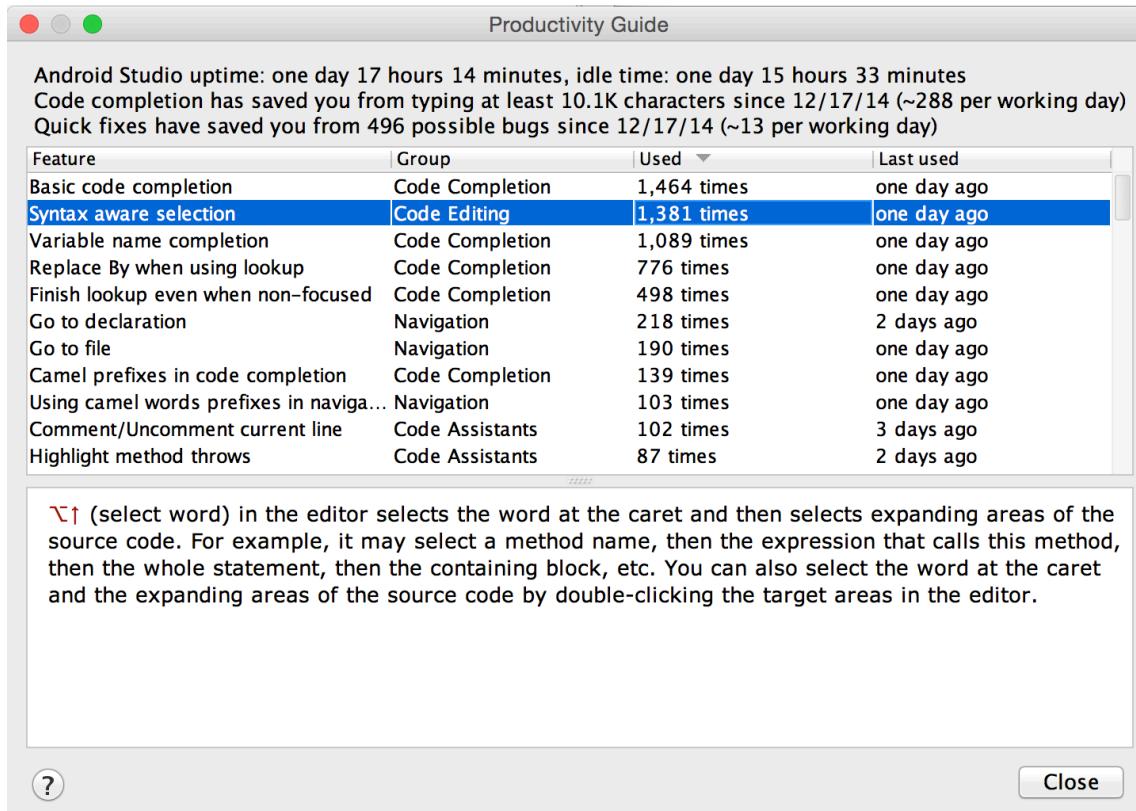
16.1 Glossary

- TDD - Test driven development
- kata - repetition to improve a specific skill

16.2 Resources

16.3 Improving Productivity

To open the Productivity Guide to see which shortcuts you use and which ones you could be using, navigate to `Help > Productivity Guide`.



This is a great way to identify ways to improve your productivity. It shows you the shortcut, how many times you've used, and the last time you used it.

To install new plugins, navigate `Android Studio > Preferences`. In the preferences dialog, search (at the top) for the word “plugins” or select `Plugins` from the list of options. Two that sound promising are “Productivity Log” and “IntelliJ Code Golf.” Finally, there’s an [older plugin](#) that may be of some use.

Check out [Android Studio Tips and Tricks](#). Another great way to search for new shortcuts is to check out IntelliJ’s documentation online. They have [key-](#)

board shortcuts by category and [shortcuts by keystroke](#).

Shortcuts

There's also a recent series that shows the shortcuts for many different scenarios. It lists the Mac, Windows, and Linux equivalents.

- [Code Generation](#)
- [Code](#)
- [Debugging Tips](#)
- [Navigation](#)
- [Navigation](#)
- [Navigation](#)
- [VCS, Debugging, Selection](#)
- [VCS, Debugging, Selection](#)

Chapter 17

Epilogue

Now that you know the basic building blocks of Android – how do you want to use this system to reach the far corners of the planet?

Do you care about teaching local children to read? Making biking on the streets safer? Curing or tracking diseases? Connecting farmers with market prices for their crops? Tracking water contaminants? Allowing people to better communicate? Or teaching the underprivileged about technology?

Whatever it is that you are passionate about, Android gives you the tools and the reach to make your aspirations a reality. You can have an impact in whatever way is meaningful to you.

Android is the perfect vehicle to impact billions of people! What are you waiting for?

Epilogues are for short endings beyond your last chapter, covering final thoughts on the topic or perhaps a glance into the future of your topic.

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 [Huajuapan 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.