jUnit

This paragraph will show you how to work with jUnit - a built-in tool designed for creating unit tests and testing the outcome of our running code. This is a tool for white-box testing, and has got access to the application source code.

Getting started

When working with jUnit, the first thing to do is create a new test project.

jUnit project

Go to the package manager -> New -> Other -> Android -> Andriod Test Project. Name the project, and choose the project, to be tested and the target Android API (it should be the same as the app's API). Please notice, that a new package has been created, whose name is the same as the project's, but with ".test" at the end. Right click on this package; go to New -> Other -> JUnit -> JUnit Test Case, then Check the "New JUnit 4 test" checkbox and set the name of your test (let's call it something like "ClassNameTest"). In the "Superclass" field, instead of "junit.framework.TestCase" or "java.lang.Object" type "android.test.ActivityInstrumentationTestCase2". When you're done, choose Finish.

The base for the test class is complete, now the project under test should be imported and the class under test instantiated.

We must now call the class we want to test in the constructor. An example is shown below.

```
public MainClassTest(){
    super("com.sample.testproject", MainActivity.class);
}
```

Two methods are needed - setUp() and tearDown() and they are called respectively before and after every test, so if any resources are needed they should be gathered and released in these methods. When testing an Activity, this is where it should be called.

```
MainActivity mActivity;

protected void setUp() throws Exception{
    super.setUp();
    mActivity = getActivity();
}
```

Having completed the whole basic setup for jUnit tests, we should then start to create each test - by common practice each test method should be called testMethodName().

Testing with jUnit

Creating tests

The first thing to know about testing an app is that all parts of it need to be tested. Each and every method should be verified and give the expected result, or expectedly fail when fed with bad arguments.

For the purpose of this article, a simple class has been created. It is called Vector2, and contains two double values (x and y). Vectors can be added, multiplied either by a scalar or by one another, and

the distance between them can be calculated.

Testing with jUnit requires knowledge of the code, so it is a good idea to familiarize yourself with the project attached to this article at this stage.

The very first thing we need to remember is the order in which the methods are run - every test is performed like this:

- setUp();
- testMethod();
- tearDown();

This is done so that if we use a field in any test, its alteration doesn't affect any other tests. It is very important to keep all tests independent from one another. We make all declarations that should be reset in there. In our case it's just an instance of Vector2.

```
protected void setUp() throws Exception {
    super.setUp();
    pVector = new Vector2(2, 2);
}
```

As for the tearDown() method, we don't do anything apart from calling super.tearDown(); we'll let the garbage collector take care of our Vector2 instance.

jUnit scans through the code of the testing class and finds all methods with the "test" prefix. In these methods, the most important methods are from the Assert class. Please read the documentation about the Assert class for further details
- http://developer.android.com/reference/junit/framework/Assert.html
One important thing to take into account is the assertEquals() method - if we overload the equals() method in our class, we won't be able to use assertEquals(), we will have to substitute with assertTrue(ourClassObject1.equals(secondObject)). In the example it looks like this:

```
public void testMultiplyByVector() {
    Vector2 mv = new Vector2(2, 2);

    Vector2 result = new Vector2(4, 4);
    assertTrue(result.equals(pVector.multiply(mv)));
    // assertEquals(result, pVector.multiply(mv));

}
```

The commented code would be the first instinct, but since these are not simple objects, two different instances (even though they have the same values) will be seen as not equal.

We can also test the UI, although in a purely "code" fashion. We want to verify, that all the items we have on the UI screen have been properly set in the .xml file (this is obvious when we run the application, but we might want to test this anyway).

To gain access to the UI elements, we need to import the R file of our project. We then call the elements like in the project:

```
Button button = (Button) mActivity.findViewById(R.id.button1);
```
This should be done in the setUp method.

And we verify that they exist:

```
public void testButtons(){
     assertNotNull(button1);
}
```
An interesting class from jUnit is the android.test.ViewAsserts class. It allows checking whether the desired view is visible, where it is compared to another etc.

```
public void testViews() {
   ViewAsserts.assertOnScreen(button1.getRootView(), button1);
   ViewAsserts.assertTopAligned(button1, button2);
}
```
We might want to verify what happens after an element has been touched. jUnit supports this via TouchUtils. This however must be verified with a flag that has to be set up in the project.

```
public void testTouch(){
   TouchUtils.clickView(this, button1);
   assertTrue(mActivity.getButton1Clicked());
}
```
If at any time we need to verify what happens if a button is pressed, we may inject this event into our test class via the sendKeys(int) class (for key codes go to the KeyEvent class here http://developer.android.com/reference/android/view/KeyEvent.html).
Other activities can also be instantiated from our test class, and run as the main activity. You may use the launchActivity(String packageName, Class class, Bundle bundle) method for this.

Please see the sample application attached to this article and the test classes associated with it.

Running tests

Having finished creating the tests, we now have to run them, which we do by right clicking on the test project -> Run as -> Android jUnitTest.

This will launch all the test cases, show which ones were successful and which ones failed, giving an overview of how the app works.

For more information on testing with jUnit, please go to http://developer.android.com/tools/testing/activity_test.html.

Robotium

Robotium is a third-party library that allows the user to create both white-box and black-box tests. It allows the user to easily interact with the whole UI of the app under test conditions.

Getting started

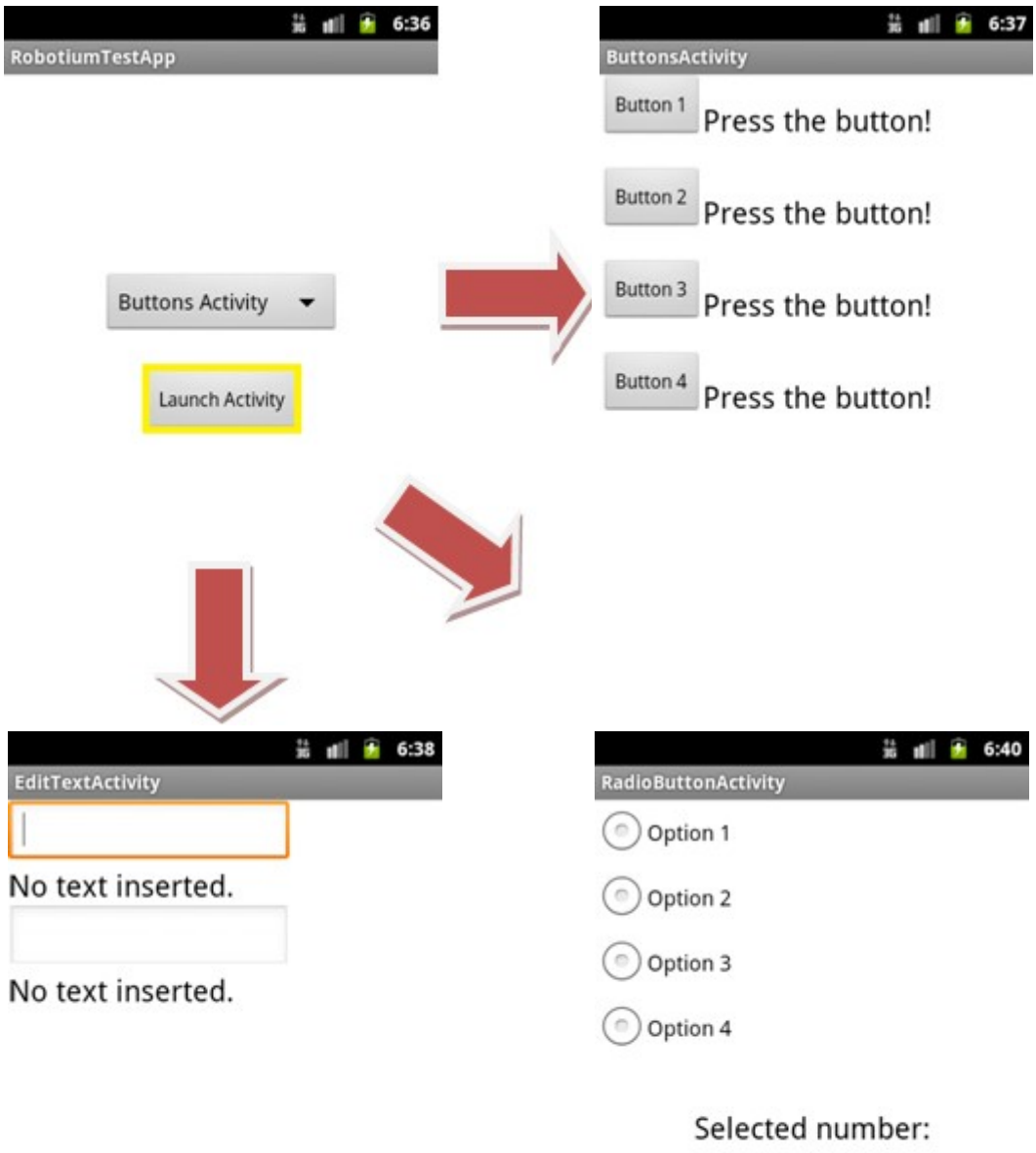
Setting up a Robotium test project is very similar to setting up a jUnit project, with the steps being identical up to the part where a test project is created. The Robotium library .jar has to be added to the project (Properties -> Java Build Path -> Libraries -> Add External JARs and then check it in Order and Export) for to be able to use it.

Since Robotium can perform black-box as well as white-box tests, we want to import both the Activity and the R class of the class under test. The constructor and the actual tests are done the same way (via the assert... method), but here's where the changes start. The main being the application interaction medium - we use the Robotium Solo class, as shown below.

```
import com.jayway.android.robotium.solo.Solo;

public class TestMan extends ActivityInstrumentationTestCase2<MainActivity> {

   private Solo solo;

   public TestMan() {
      super("com.samsung.sampletestingclass", MainActivity.class);
   }

   protected void setUp() throws Exception {
      super.setUp();
      solo = new Solo(getInstrumentation(), getActivity());
   }

   protected void tearDown() throws Exception {
      solo.finishOpenedActivities();
      super.tearDown();
   }
}
```

With Robotium we can test the same things as with jUnit, but we can also do much more - we can create test cases that interact directly with the UI and verify the correctness of the app.

For this example let's take an app with a few activities, called from one main activity. As seen on the image below we have a main screen with a spinner and a button. We navigate through the other activities by choosing one from the spinner and pressing the "Launch Activity" button.



The app is divided into three sections:

1.The 4 buttons section, after clicking each one a toast appears with the number of the pressed button

2.A part with 2 edit fields and 2 text views, upon inserting text into the edit fields the text views change their text to the one in the edit field.

3.A part with 4 radio buttons and a text view, upon clicking on a radio button, the text view tells which one was selected.

Testing with Robotium

Writing tests

As with the jUnit tool, we can perform white-box tests which will work just as with jUnit. What is really interesting is how Robotium lets the user make automated black-box tests of the UI.

The instance of the Solo class is how we connect to the app and through it, we can inject any event we want.

Elements of the UI, like buttons for example can be accessed in two ways - by their text or by their index.

```
solo.clickOnButton("Press me!");
solo.clickOnButton(0);
```

The index values are set up by the order in which they have been created in the layout file, so for example the code above will click on a button with the text “Press me!” on it (if it can find one, if not the test will return failed) and then the first button designed in the XML file.

> if buttons are overlapping, this can lead to not all the buttons being clicked - only the top one (to which there is access) will be touched.

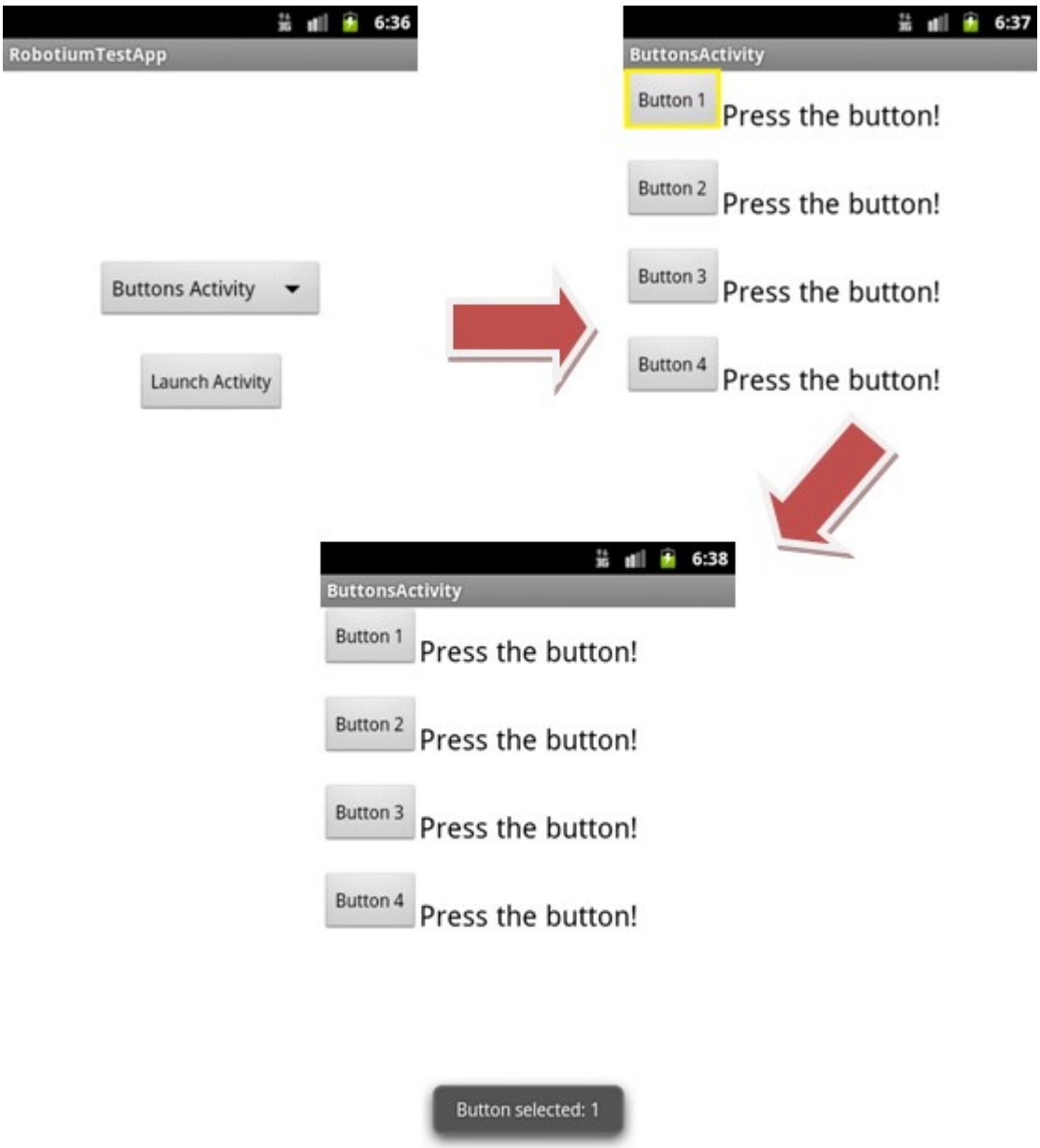
Robotium also provides the means to verify whether the events had the desired effect - we can search for text on screen with

```
solo.searchText(solo.getString(R.string.Toast_text));
```

which can even find text on toasts.

We want to test all the functions of the test app, so we will make a test for each button, edit field and radio button.

Here is what we want to achieve:



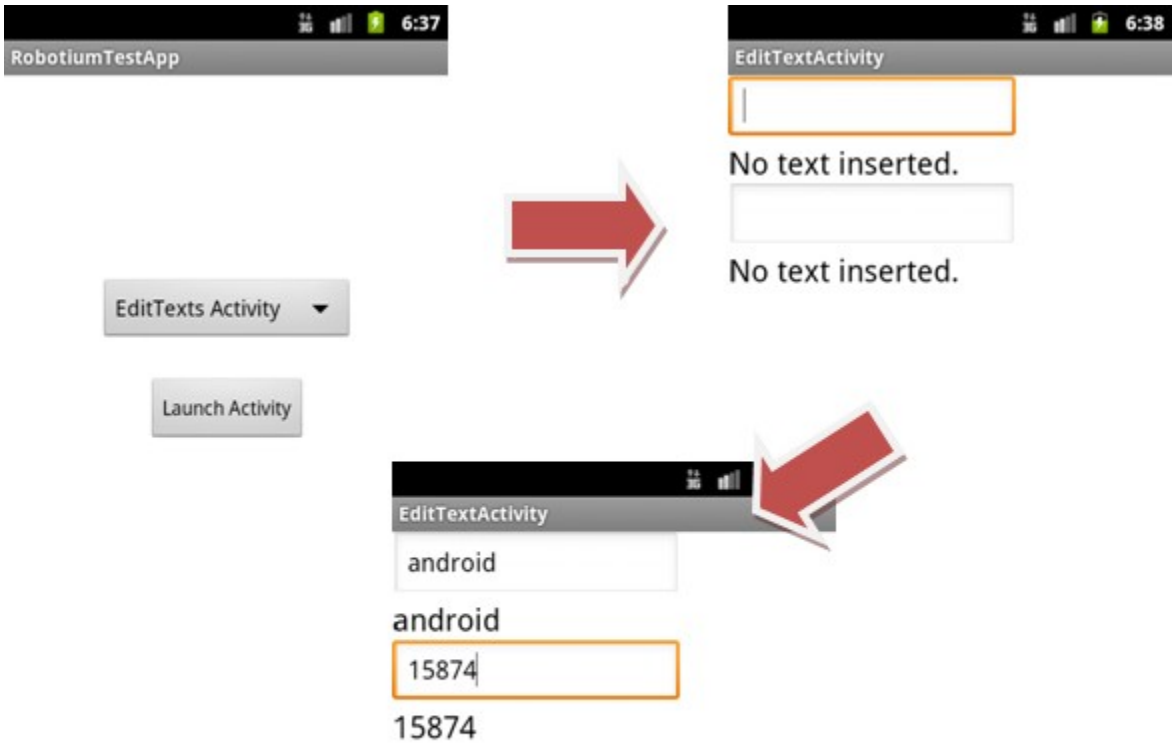
So we want to go to the first app, press the first button and search for the text “Button selected: 1”.

```
public void testButtons1() {
    solo.pressSpinnerItem(0, 0);
    solo.clickOnButton(0);
    solo.assertCurrentActivity("Buttons Activity", ButtonsActivity.class);
    solo.clickOnButton("Button 1");
    solo.searchText("Button selected: 1");
}
```

At first we are in the main class, we navigate through the spinner and launch the second activity. Once there, we press the button marked “Button 1” and search for the desired text. As mentioned before, this could also be achieved by choosing the button by its ID and not by its text.

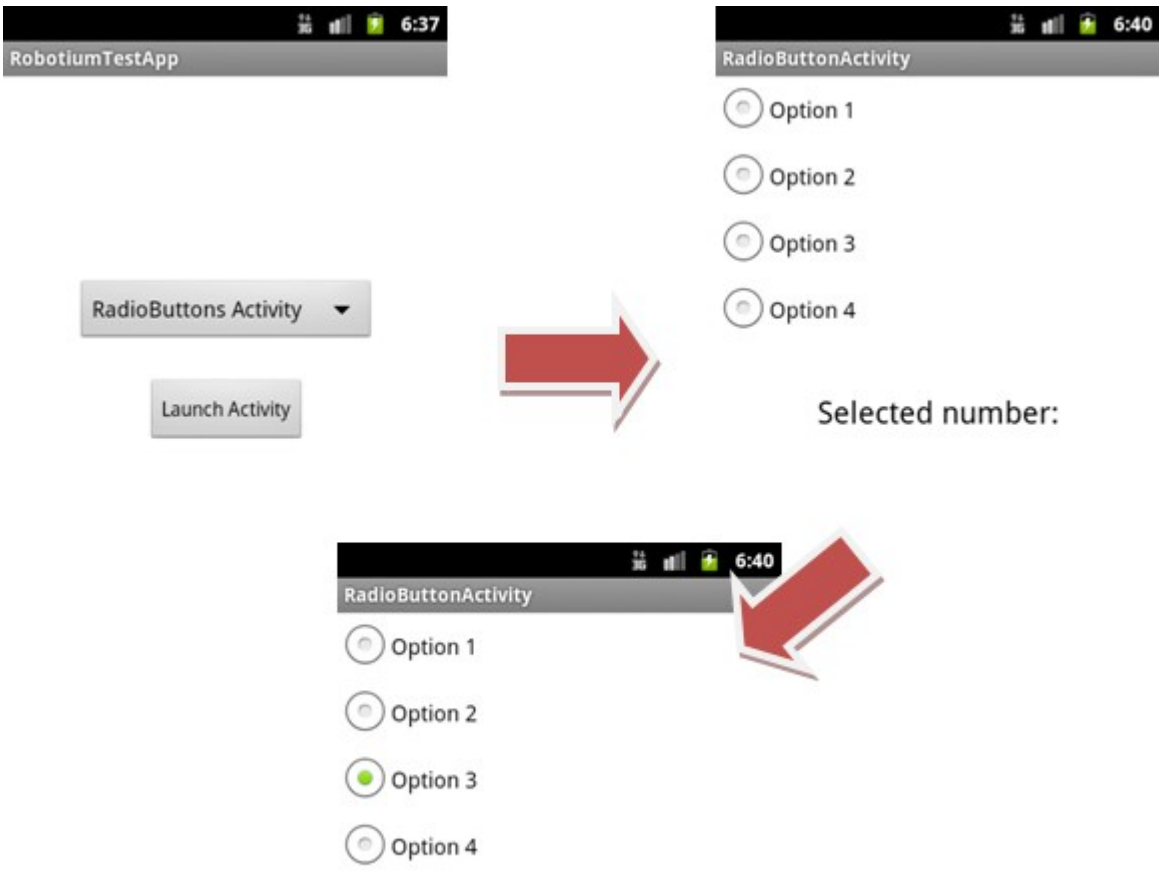
```
public void testButtons2() {
    goToButtons();
    solo.assertCurrentActivity("Buttons Activity", ButtonsActivity.class);
    solo.clickOnButton(0);
    solo.searchText("Button selected: 1");
}
```

The goToButtons() method seen above is a method that brings us to the Buttons activity. A similar one has been written for the other activities.




```
public void testEditText2() {
goToEditTexts();
    solo.assertCurrentActivity("EditText Activity", EditTextActivity.class);
    String text = "this is a new text";
    solo.typeText(solo.getEditText(0), text);
    solo.searchText(text);
}
```

The editText can also be accessed by the text it has, but since this is an editable field, this is not recommended.




```
public void testRadioButtons1() {
    goToRadioButtons();
    solo.assertCurrentActivity("Radio Buttons Activity", RadioButtonActivity.class);
    solo.clickOnRadioButton(0);
    solo.searchText(solo.getString(R.string.radio_textView) + " 1");
}
```

In the snippet above we don’t search for a text that has been added in the test code, but one that has been included in the class. We first imported the R class, so we can reach it with our solo instance.

The tests are run in the same way as jUnit ones.

Another interesting fact about this tool is that we can test apps without having access to their code - all that is needed is the package name and main activity name, which can be obtained by running the app and connecting the device to Logcat - they will be shown by the ActivityManager.

These tests are all attached in the sample test app attached to this article.

Monkey tool

Monkey is a tool for black-box testing, which can be seen as two parts - the Monkey and the MonkeyRunner.

Monkey

Monkey is a command line tool, designed to randomly test a desired app or device. It will bombard the device with events generated with a specified delay, or as quickly as possible. All the options are talked about in fine detail at http://developer.android.com/tools/help/monkey.html.

MonkeyRunner

The MonkeyRunner is a more sophisticated Monkey. It allows creating scripted tasks to test the application, however knowledge of Python is needed.

The tool is comprised of 3 classes - MonkeyRunner, which can interface with the user, MonkeyDevice, which has all the interesting input methods and MonkeyImage, a class designed specifically to get images, compare images etc.

Scripts are created to test specific applications - the package name and activity name are required.

A basic script looks something like this:

```
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice


monkeyDevice = MonkeyRunner.waitForConnection()
monkeyDevice.installPackage('bin/SampleTestingClass.apk')
package = 'com.samsung.sampletestingclass'
activity = 'com.samsung.sampletestingclass.MainActivity'
runComponent = package + '/' + activity
monkeyDevice.startActivity(component=runComponent)
```

What has happened here is a MonkeyDevice instance is created and attached to the current device (emulator or physical). The .apk is installed on the device and the activity is run.

We can now write scenarios that will be run on the working app.

The main methods we'll be using for this are in the MonkeyDevice class.

Unfortunately this tool doesn't have the means to access the resources of the class that is under test, so we can't for example make it press a specific button on the UI just by giving its name. We have to navigate through all the UI elements with DPAD_UP, DPAD_DOWN, DPAD_LEFT and DPAD_RIGHT commands

```
monkeyDevice.press("DPAD_DOWN", MonkeyDevice.DOWN_AND_UP)
monkeyDevice.press("DPAD_CENTER", MonkeyDevice.DOWN_AND_UP)
```

and press the desired element with DPAD_CENTER.

We could also send a touch command and supply the x,y coordinates

```
monkeyDevice.touch(x, y, MonkeyDevice.DOWN_AND_UP)
```

This tool can also insert strings, take screenshots (and compare them!) and show all the information on the console.

A short sample Python script has been added to this article for your testing pleasure.

Ref:http://developer.samsung.com/android/technical-docs/Testing-the-UI-and-Activity