Unit Testing in Android Studio

Terry Carroll
2019-09-27

What is unit testing?

Wikipedia: “In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.”

Unit testing is a way of testing program behavior at the method level to ensure the pieces act the way you want them to act.

Unit testing is *not*:

- Integration testing (confirming that individual units work well together); or
- System testing (ensuring that overall system works as designed).

Benefits of unit testing

- Testing improves code quality.
- Bugs are found early.
    - By testing as you code, bugs are found as code is created.
- Bugs don’t creep [back] in.
    - You keep the tests, so every time you change the code they get re-run.
- Iterative improvement and assurance when debugging.
    - You continually add tests for what’s needed as development ensues.

Side benefits of unit testing (besides code quality)

- Design improvements:
  - Makes you think through the problem before you begin coding;
  - Tends toward designing the program in more granular units.
- Facilitates division of labor:
  - Smaller units means it’s easier to divide up the work;
  - Unit test writing can be done separately from implementation coding.
- You’re probably doing most of the work anyway; taking a unit-testing approach saves that work for reuse and quality assurance.
- Will probably look good for the engineering notebook.

Example: isLeapYear(*int year*)

Most years are not leap years, but...

> years divisible by 4 are leap years, except…
>
> > years divisible by 100 are *not* leap years, even though they're divisible by 4, except…
> >
> > > years divisible by 400 are leap years after all, even though they're divisible by 100.

One more thing: this is the *Gregorian* calendar, not adopted in England (and its American colonies) until mid-1752; so any year prior to 1752 is not a leap year (or, more accurately, did not have a February 29).

Warning: don’t really write code like this

- The code I’ll show is just for illustrative purposes.
- In the real world, don’t write code to do things like figure out whether a year is a leap year; use a library if available:

```
boolean its_a_leap_year = java.time.Year.of(year).isLeap();
```

or even:

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.YEAR, year);
boolean its_a_leap_year = cal.getActualMaximum(Calendar.DAY_OF_YEAR) > 365;
```

Sample unit tests for isLeapYear()

```
@Test
public void testTypicalYears(){
        assertTrue(MainActivity.isLeapYear(2016));
        assertFalse(MainActivity.isLeapYear(2017));
        assertFalse(MainActivity.isLeapYear(2018));
        assertFalse(MainActivity.isLeapYear(2019));
        assertTrue(MainActivity.isLeapYear(2020));
        assertFalse(MainActivity.isLeapYear(2021));
        assertFalse(MainActivity.isLeapYear(2022));
        assertFalse(MainActivity.isLeapYear(2023));
        }
```

Sample unit tests for isLeapYear()

```
@Test
public void testCenturies(){
        assertTrue(MainActivity.isLeapYear(1600));
        assertFalse(MainActivity.isLeapYear(1700));
        assertFalse(MainActivity.isLeapYear(1800));
        assertFalse(MainActivity.isLeapYear(1900));
        assertTrue(MainActivity.isLeapYear(2000));
        assertFalse(MainActivity.isLeapYear(2100));
        assertFalse(MainActivity.isLeapYear(2200));
        assertFalse(MainActivity.isLeapYear(2300));
        assertTrue(MainActivity.isLeapYear(2400));
        }
```

Implementation; first cut

```
public static boolean isLeapYear(int year){
        boolean result;
        if (year % 4 == 0) {
                result = true;
                }
        else {
                result=false;
                }
        return result;
        }
```

These tests all work

```
@Test
public void testTypicalYears(){
	assertTrue(MainActivity.isLeapYear(2016));
	assertFalse(MainActivity.isLeapYear(2017));
	assertFalse(MainActivity.isLeapYear(2018));
	assertFalse(MainActivity.isLeapYear(2019));
	assertTrue(MainActivity.isLeapYear(2020));
	assertFalse(MainActivity.isLeapYear(2021));
	assertFalse(MainActivity.isLeapYear(2022));
	assertFalse(MainActivity.isLeapYear(2023));
	}
```

These, not so much

```
@Test
public void testCenturies(){
        assertTrue(MainActivity.isLeapYear(1600));
        assertFalse(MainActivity.isLeapYear(1700)); ← fail
        assertFalse(MainActivity.isLeapYear(1800)); ← fail
        assertFalse(MainActivity.isLeapYear(1900)); ← fail
        assertTrue(MainActivity.isLeapYear(2000));
        assertFalse(MainActivity.isLeapYear(2100)); ← fail
        assertFalse(MainActivity.isLeapYear(2200)); ← fail
        assertFalse(MainActivity.isLeapYear(2300)); ← fail
        assertTrue(MainActivity.isLeapYear(2400));
        }
```

Implementation; second cut

```
public static boolean isLeapYear(int year){
        boolean result;
        if (year % 100 == 0){
                result = false;
                }
        else (year % 4 == 0) {
                result = true;
                }
        else {
                result=false;
                }
        return result;
        }
```

Still some failures, but better

```
@Test
public void testCenturies(){
        assertTrue(MainActivity.isLeapYear(1600)); ← fail
        assertFalse(MainActivity.isLeapYear(1700));
        assertFalse(MainActivity.isLeapYear(1800));
        assertFalse(MainActivity.isLeapYear(1900));
        assertTrue(MainActivity.isLeapYear(2000)); ← fail
        assertFalse(MainActivity.isLeapYear(2100));
        assertFalse(MainActivity.isLeapYear(2200));
        assertFalse(MainActivity.isLeapYear(2300));
        assertTrue(MainActivity.isLeapYear(2400)); ← fail
        }
```

Implementation; reasonable working version (all test calls, in both tests, pass)

```
public static boolean isLeapYear(int year){
        boolean result;
        if (year % 400 == 0) {
                result = true;
                }
        else if (year % 100 == 0){
                result = false;
                }
        else if (year % 4 == 0){
                result = true;
                }
        else {
                result=false;
                }
        return result;
        }
```

What about pre-Gregorian years (before 1753)?

Let’s add some code to make all pre-Gregorian years return as non-leap year.

Pre-Gregorian support, first try

```
public static boolean isLeapYear(int year){
        boolean result;
        if (year >= 1753){
                result = true;
                }
        else if (year % 400 == 0) {
                result = true;
                }
        else if (year % 100 == 0){
                result = false;
                }
        else if (year % 4 == 0){
                result = true;
                }
        else {
                result=false;
                }
        return result;
        }
```

Unit test for pre-Gregorian

```
@Test
public void testPreGregorian(){
   assertFalse(MainActivity.isLeapYear(1750));
   assertFalse(MainActivity.isLeapYear(1751));
   assertFalse(MainActivity.isLeapYear(1752));
   assertFalse(MainActivity.isLeapYear(1753));
   assertFalse(MainActivity.isLeapYear(1754));
   assertFalse(MainActivity.isLeapYear(1755));
   assertTrue(MainActivity.isLeapYear(1756));   /* First Gregorian leap year*/
   assertFalse(MainActivity.isLeapYear(1757));
   assertFalse(MainActivity.isLeapYear(1758));
   assertFalse(MainActivity.isLeapYear(1759));
   assertTrue(MainActivity.isLeapYear(1760));
   }
```

Pre-Gregorian support, first try: fail

Not only does it not work for pre-1753 years; but it reports every post-1753 year as a leap year.

As bugs go, this is not so bad. You'd find it didn't work for pre-1753 right away; but if the bug were one that *worked* for pre-1753, but *broke* for post-1753, you might not think of testing the post-1753 code.

Unit testing will catch this.

Pre-Gregorian support, corrected

```
public static boolean isLeapYear(int year){
        boolean result;
        if (year < 1753){
                result = false;
                }
        else if (year % 400 == 0) {
                result = true;
                }
        else if (year % 100 == 0){
                result = false;
                }
        else if (year % 4 == 0){
                result = true;
                }
        else {
                result=false;
                }
        return result;
        }
```

Old tests need to be revisited of expected performance fails

Remember this?

```
assertTrue(MainActivity.isLeapYear(1600));
```

Now that we consider all pre-1753 years to be non-leap years, that test fails and needs to be updated to reflect our new criteria.

```
assertFalse(MainActivity.isLeapYear(1600));
```

What about non-positive integer years?

Current code says year 0 -- and for that matter years -4, -8, -16, etc. -- are leap years!

And what about years -1, -2019, etc.?

It’s wrong to say they’re leap years, and wrong to say they’re not leap years. They’re not valid years at all.

What about non-positive integer years?

```
public static boolean isLeapYear(int year){
        boolean result;
        if (year < 1){
                throw new IllegalArgumentException(String.format("Invalid year value '%s'.", year));
                }


        if (year < 1753){
                result = false;
                }

                . . .

        return result;
```

Testing for exceptions

```
@Test(expected=IllegalArgumentException.class)
public void testBadYearZero(){
     MainActivity.isLeapYear(0);
}

@Test(expected=IllegalArgumentException.class)
public void testBadYearNegative(){
     MainActivity.isLeapYear(-1);
}
```

Note: best to keep these as separate tests; otherwise you're only ensuring that the test method will throw the exception, i.e., that *at least one of* the `isLeapYear()` calls will throw the exception; the "`expected=`" clause applies to the entire test method.

assertEquals / assertNotEquals

```
public static int adder(int a, int b){
	return a + b;
}

@Test
public void testAdder(){
	assertEquals(MainActivity.adder(32, 10), 42);
	assertNotEquals(MainActivity.adder(64, -12), 42);
	}
```

assertEquals / assertNotEquals
For float & double, use acceptable delta value

```
public static double lame_sqrt(double x){
   int number_of_iterations = 10;
   double guess = 2.0;
   double approximate_root = 0;
   for (int i = 1; i <= number_of_iterations; i++){
      approximate_root = x/guess;
      guess = (approximate_root + guess)/2.0;
      }
    return approximate_root;
    }

@Test
public void testSqrt(){
    assertEquals(MainActivity.lame_sqrt(9.0), 3.0, 0.0001);
    assertEquals(MainActivity.lame_sqrt(10000.0), 100.0, 0.0001);
    assertEquals(MainActivity.lame_sqrt(2.0), 1.414, 0.001);
    }
```

A test method does not have to just be a list of asserts

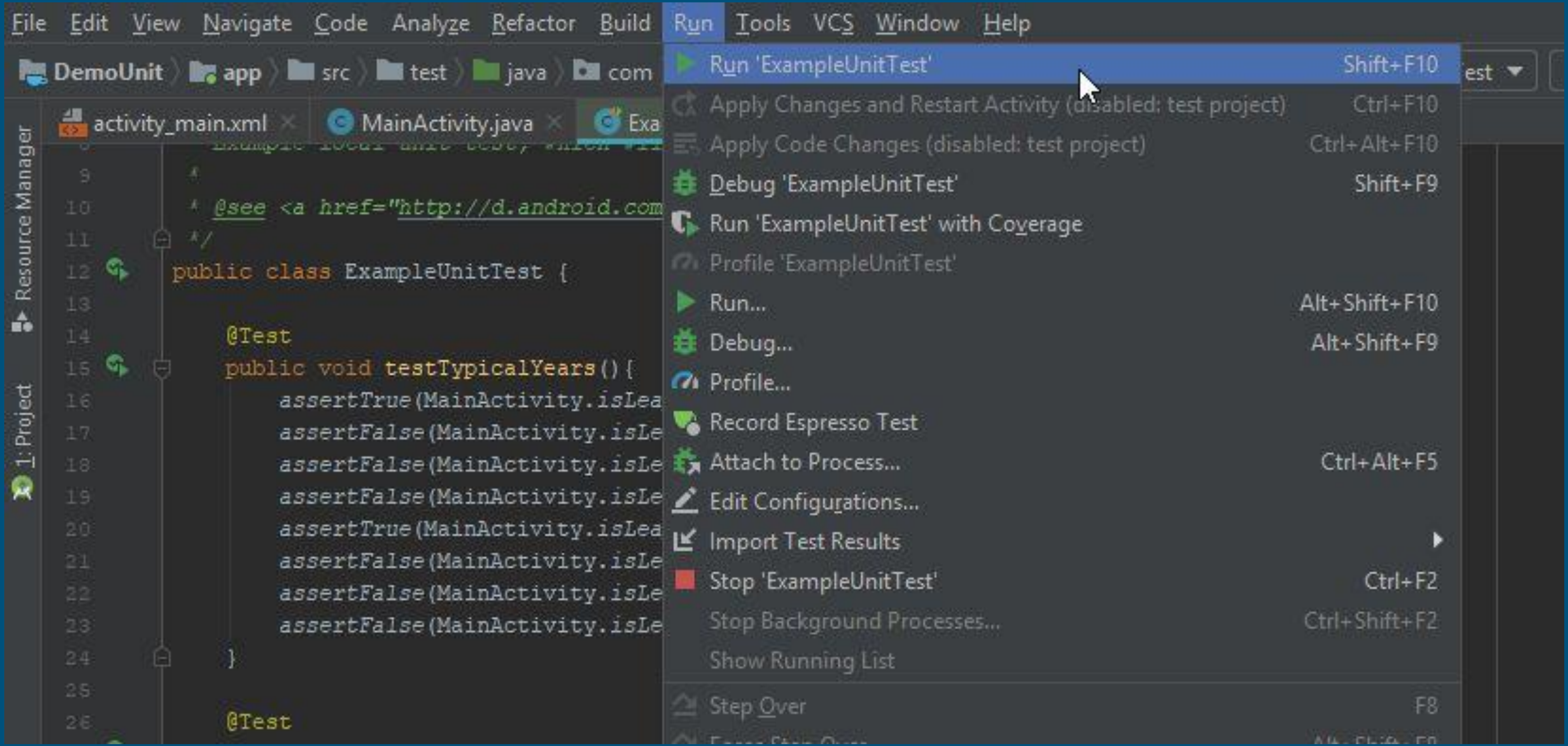
Say you have a method isValidMonth() that checks to see if the month number is valid.

```
@Test
public void testValidMonth(){
     int[] good_months_to_test = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
     int[] bad_months_to_test = {0, 13, -1, 100, 42, 365};
     for (int month : good_months_to_test){
         assertTrue(MainActivity.isValidMonth(month));
          }
     for (int month : bad_months_to_test){
         assertFalse(MainActivity.isValidMonth(month));
          }
     }
```

Android Studio

- `View` / `Tool Windows` / `Project` to get the project view
- Unit test class is in `app/java` folder, marked “(test)”, probably named `ExampleUnitTest`.
    - Not “(instrumented test)” or “(AndroidTest)”
- I needed to add: `import static org.junit.Assert.*;` to get the Assertion methods.
- To run the tests:
    - `Run` / `Run 'ExampleUnitTest'` (or other name of the test class); or
    - Right-click on the test class in the Project view and select `Run 'ExampleUnitTest'`.

Running unit tests in Android Studio



```
 *
 * @see <a href="http://d.android.com
 */
public class ExampleUnitTest {

    @Test
    public void testTypicalYears(){
        assertTrue(MainActivity.isLea
        assertFalse(MainActivity.isLe
        assertFalse(MainActivity.isLe
        assertFalse(MainActivity.isLe
        assertTrue(MainActivity.isLea
        assertFalse(MainActivity.isLe
        assertFalse(MainActivity.isLe
        assertFalse(MainActivity.isLe
    }

    @Test
```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
DemoUnit app src test java com
activity_main.xml MainActivity.java Exa
Resource Manager
1: Project
Run 'ExampleUnitTest' Shift+F10
Apply Changes and Restart Activity (disabled: test project) Ctrl+F10
Apply Code Changes (disabled: test project) Ctrl+Alt+F10
Debug 'ExampleUnitTest' Shift+F9
Run 'ExampleUnitTest' with Coverage
Profile 'ExampleUnitTest'
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Profile...
Record Espresso Test
Attach to Process... Ctrl+Alt+F5
Edit Configurations...
Import Test Results
Stop 'ExampleUnitTest' Ctrl+F2
Stop Background Processes... Ctrl+Shift+F2
Show Running List
Step Over F8

Tips

- Isolate robot input (such as sensors) and output (such as control) into their own methods; and everything before and after into their own methods. That way you can more easily simulate and test all the parts that don't actually talk to hardware.
- More sophisticated testing tactics (only as needed):
  - `setUp()` before tests; `tearDown()` after (for example to load image files)
  - Mock objects (Mockito?)
  - (I haven't done either of these in Java.)
- It's easy to overdo it. The examples here and in the code repository *do* overdo it, for the sake of illustration. Don't let coding tests get in the way of coding useful stuff.

Other info

- Pretty much every language has a unit test framework, or more than one; and pretty much every IDE has unit test support. xUnit (JUnit, NUnit, etc.) is the most popular and best-documented.
- Unit test frameworks vary by language and platform; what we looked at here is JUnit under Android Studio.
- Sometimes what works varies from what you find googling, because of a different environment or release.
- Only public methods are accessible to unit test.

URLs

Unit testing in Android Studio: https://developer.android.com/training/testing/unit-testing

Decent tutorial: http://tutorials.jenkov.com/java-unit-testing/index.html

My sample code repository: https://github.com/codingatty/DemoUnit
Code:
https://github.com/codingatty/DemoUnit/blob/master/app/src/main/java/com/example/demounit/MainActivity.java
Tests:
https://github.com/codingatty/DemoUnit/blob/master/app/src/test/java/com/example/demounit/ExampleUnitTest.java