Engineering Design w/Embedded Systems

Lecture 16—Testing; Writing JUnit Tests

Patrick Lam
University of Waterloo

February 7, 2013

Part I

The JUnit Unit Testing Framework

Unit Testing

Units: small parts of a software system (methods, classes, sets of classes).

Two Key Ideas:

1. Test units independently.
2. Specify desired behaviour using tests.

About JUnit

Most popular unit testing framework for Java.

Assignment 6: develop standalone JUnit tests as well as Android JUnit tests.

Tests depend on the notion of **assertions**.

Objective: Unit Tests

Learn how to write simple JUnit tests for Java code.

Use `assertTrue` or `assertEquals` to verify test results.

JUnit Test Organization

JUnit tests belong to test classes.

A test:

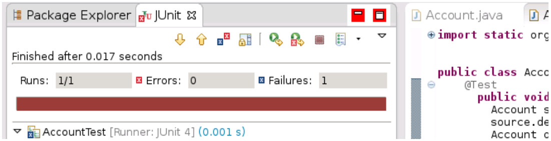
- is labelled `@org.junit.Test`;
- is a method with no parameters;
- makes calls to the class under test;
- verifies that the class is doing the right thing using assertions.

Using JUnit Tests

After writing the tests:

1. press a button in your IDE;
2. it will run the tests automatically for you.

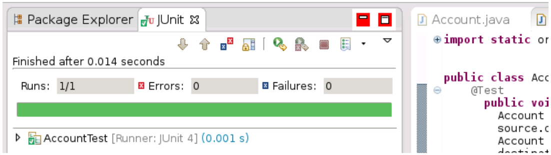
If the tests fail:



```
Finished after 0.017 seconds
Runs: 1/1    Errors: 0    Failures: 1
AccountTest [Runner: JUnit 4] (0.001 s)
```

```
import static org
public class Acco
    @Test
        public void
            Account s
            source.de
            Account c
```

If the tests pass:



```
Finished after 0.014 seconds
Runs: 1/1    Errors: 0    Failures: 0
AccountTest [Runner: JUnit 4] (0.001 s)
```

```
import static or
public class Acc
    @Test
        public voi
            Account
            source.c
            Account
```

Why Unit Tests?

Biggest benefit of unit tests:

- they can run automatically.

You'll never run tests that are annoying to run.

Account Example

```
public class Account
{
  private float balance;
  public void deposit (float amount) { balance += amount; }

  public void withdraw (float amount) { balance -= amount; }

  public void transferFunds(Account destination, float amount) {
  }

  public float getBalance() { return balance; }
}
```

JUnit Test for Account

Let's write a unit test for this class:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class AccountTest {
  @Test
  public void transferFunds() {
    Account source = new Account();
    source.deposit(200.00f);
    Account destination = new Account();
    destination.deposit(150.00f);

    source.transferFunds(destination, 100.00f);
    assertEquals("destination balance", 250.00f,
                 destination.getBalance(), 0.01);
    assertEquals("source balance", 100.00f,
                 source.getBalance(), 0.01);
  }
}
```

Test Results

If we run this test, we'd get a red bar; `transferFunds` doesn't do anything. Adding this code:

```
public void transferFunds(Account destination, float amount) {
    destination.deposit(amount);
    withdraw(amount);
}
```

makes the bar green, since the test now passes.

JUnit Test Fixtures

Tests often share objects.

Repeatedly allocating these objects is terrible.

Solution: test fixtures.

```
public class AccountTest {
  private Account account1;

  // runs before any tests
  @Before public void setUp() {
    account1 = new Account();
    account1.deposit(200.00f);
  }
}
```

Use `@After` to free resources afterwards.

Caveat: Don’t rely on object state between tests: JUnit may shuffle order.

Expected Exceptions

How do you test error-checking code?

It should throw exceptions!

Make the test case expect an exception.

From the JUnit cookbook:

```
@Test(expected=IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

Expected Exceptions

How do you test error-checking code?

It should throw exceptions!

Make the test case expect an exception.

From the JUnit cookbook:

```
@Test(expected=IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

JUnit 4 versus JUnit 3

So far, we saw the (better) JUnit 4 syntax.

Android uses JUnit 3 syntax. Key differences:

- test classes must extend `TestCase`;
- test names must start with `test`;
- fixture setup method must be called `setUp()`;
- fixture teardown method must be called `tearDown()`;
- testing exceptions is harder.