JUnit Testing

JUnit testing framework, syntax, usage, best practices

What is JUnit?

- A **unit test** is a test of a single unit of functionality of code (almost always a single method).
- **JUnit** is a framework for creating and executing unit tests:
  - part of the Java API.
  - Automates "white box testing."
- A JUnit **test case** is a class that defines unit tests to be executed. Each JUnit test case focuses on a single class to be tested.
- A JUnit **test method** is a method in a JUnit test case designed to test the code in question.
- You should focus on testing paths through methods of the class under test.
- There are different versions of Junit. These slides use JUnit 4.
- Need to include JUnit libraries on classpath. (Eclipse does this for you.)

Naming conventions

- `MyClass` is a class: `MyClassTest` is the name of its test case.
- `myMethod` is a method: `testMyMethod` (or variation thereon) is the name of its test method.

ClockCalculator class to test

```
public class ClockCalculator {
   private int time = 12;

   public ClockCalculator() { }
   public ClockCalculator(int time) {
      this.time = Math.abs(time)%12;
      setTo12();
   }
   private void setTo12() { if (time == 0) time = 12; }
   public int getTime() { return time; }
   public void add(int x) { time += x; }          // Wrong

   public void multiply(int x) {
      time = (x % 12) * (time % 12);        // Wrong
      setTo12(); }
   public void divide(int x) {
      time = time / x % 12;  //Wrong, can generate an error
      setTo12(); }
}
```

ClockCalculator test case

```
import static org.junit.Assert.*;
import org.junit.*;
public class ClockCalculatorTest {

   @Before
   public void setUp() throws Exception {    }
   @Test
   public void testClockCalculator() {
      fail("Not yet implemented");
   }
   @Test
   public void testClockCalculatorInt() {
      fail("Not yet implemented");
   }
   @Test
   public void testAdd() {
       fail("Not yet implemented"); }
   // … more test methods
}
```

Test case imports, annotations

Imports

- `org.junit.*` (covers `org.junit.Before` & `org.junit.Test`)
- `static org.junit.Assert.*`
  [static means you don't have to specify the class where the method is defined]

Annotations:

- provide data about a program – for the compiler, tool processing, runtime processing.
- Examples: `@Deprecated, @Override, @Test`
- `@Before` – tells JUnit test runner that this method is to be run prior to every test method execution
- `@Test` – tells JUnit test runner that this is a test method

@Before and setUp( )

`setUp()`:

- Used typically to initialize an object.
- executed immediately prior to execution of every test method.
- `@Before` annotation tells JUnit test runner that this is your `setUp()` method.

```
public class ClockCalculatorTest {
    private ClockCalculator cc;

    @Before
    public void setUp() throws Exception {
        cc = new ClockCalculator();
    }
```

Asserts

- Assert methods generate output only when there is a failure
- `assertEquals()`:
  - `assertEquals(int expected, int actual)`

  -
    ```
    assertEquals(double expected,
                 double actual, double tolerance)
    ```
  - `assertEquals(Object expected, Object actual)`
- `assertTrue(boolean actual)`
- `assertFalse(boolean actual)`
- `assertNull(Object actual)`
- `assertNotNull(Object actual)`
- `fail()` always generates a failure.

All of these methods have an optional leading String parameter containing a message. Example:

```
assertEquals("Initialized hour to 13", 1, cc.getTime());
```

Writing test methods

1. Decide which methods to test.
2. Get eclipse to generate stubs for each method to be tested.
3. (Usually) declare an instance variable of the type of class under test.
4. Create setUp().
5. For each test method, call the method that you are testing.
6. Write an assert method call to determine if the results are correct.
7. Repeat 5 and 6 as needed to test the method. If you have multiple tests in a single test method, use the String parameter to indicate exactly what you are testing.

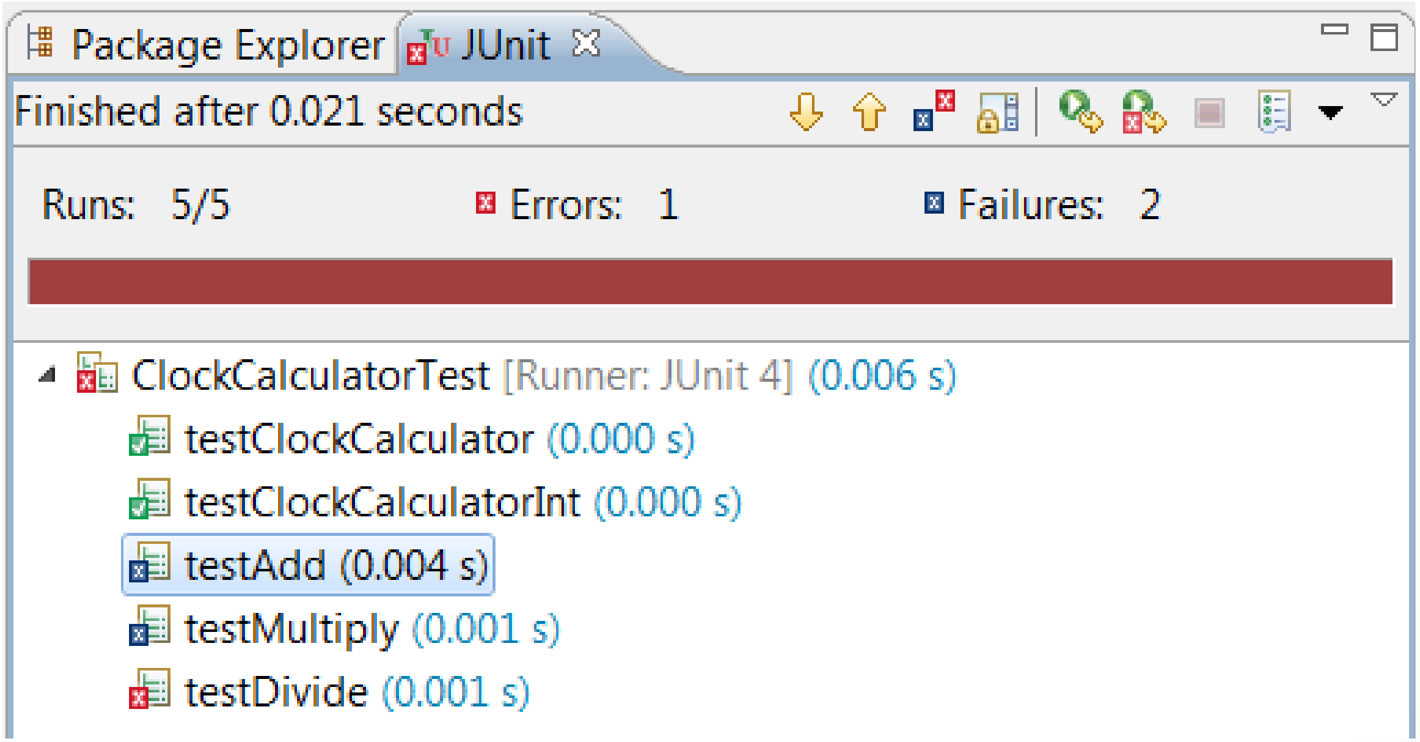
Asserts in ClockCalculatorTest

```
    @Test
    public void testClockCalculator() {
       assertEquals(12, cc.getTime());
    }

    @Test
    public void testClockCalculatorInt() {
       cc = new ClockCalculator(13);
       assertEquals("Initialize to 13", 1, cc.getTime());
       cc = new ClockCalculator(-5);
       assertEquals("Initialize to -5", 12, cc.getTime());
    }
```

JUnit test results

- **Error**. The test method generated a runtime error.
- **Fail**. The assert in the test method failed but there was no runtime error.
- **Pass**. The test method did not generate a runtime error or a failure.



Package Explorer | JUnit

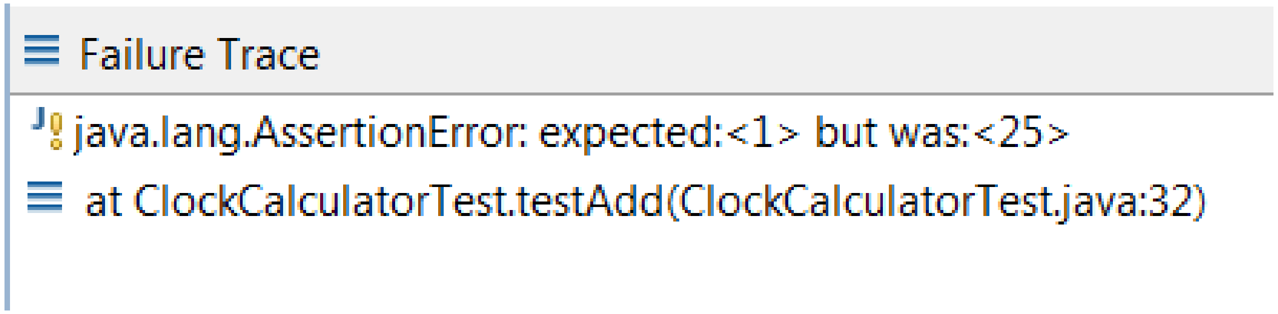
Finished after 0.021 seconds

Runs: 5/5 Errors: 1 Failures: 2

ClockCalculatorTest [Runner: JUnit 4] (0.006 s)
- testClockCalculator (0.000 s)
- testClockCalculatorInt (0.000 s)
- testAdd (0.004 s)
- testMultiply (0.001 s)
- testDivide (0.001 s)

Failure test results

```
@Test
public void testAdd() {
   cc.add(13);
   assertEquals(1, cc.getTime());
}
```

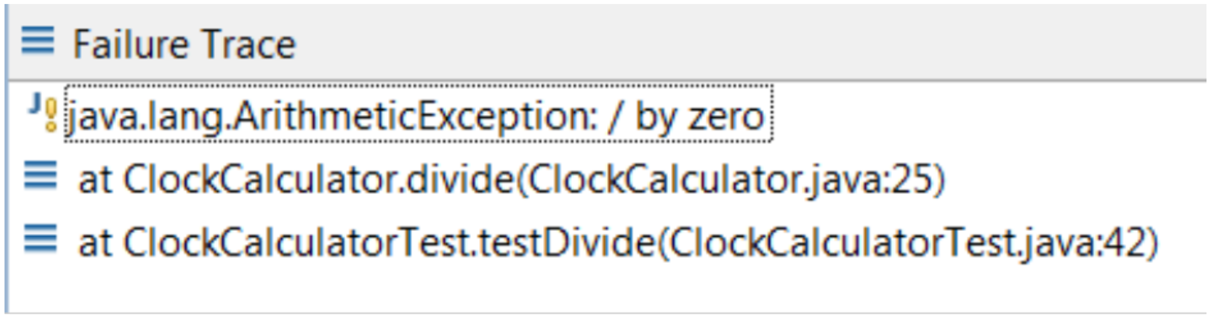


```
Failure Trace
java.lang.AssertionError: expected:<1> but was:<25>
at ClockCalculatorTest.testAdd(ClockCalculatorTest.java:32)
```

```
java.lang.AssertionError: expected:<1> but was:<25>
  at org.junit.Assert.fail(Assert.java:91)
…
  at ClockCalculatorTest.testAdd(ClockCalculatorTest.java:32
```

Error test results

```
@Test
public void testDivide() {
   cc.divide(0);
   assertEquals("Divide by 0", 12, cc.getTime());
}
```



Failure Trace

```
java.lang.ArithmeticException: / by zero
at ClockCalculator.divide(ClockCalculator.java:25)
at ClockCalculatorTest.testDivide(ClockCalculatorTest.java:42)
```

```
java.lang.ArithmeticException: / by zero
        at ClockCalculator.divide(ClockCalculator.java:25)
        at ClockCalculatorTest.testDivide(ClockCalculatorTest.java:42)
…
```

JUnit test strategies

- Create a test class for each class you need to test.
- Create a test method for each public method in a class that could possibly fail.
- Each individual set of test data requires at least one assert statement.
- For each method under test, consider each path through the method and create test data :
    - Try to cover as many different paths of execution as feasible. Test the critical paths. Create multiple test methods for complicated paths.
    - At decision points, test with true *and* false values.
    - Test with boundary value data (at the ends of acceptable ranges).
    - Test values in each equivalence class of possible input.
    - For collections, always test empty collections and (if possible) full ones.