Software Testing and Reliability

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

Lecture 11

JUnit

Unit testing

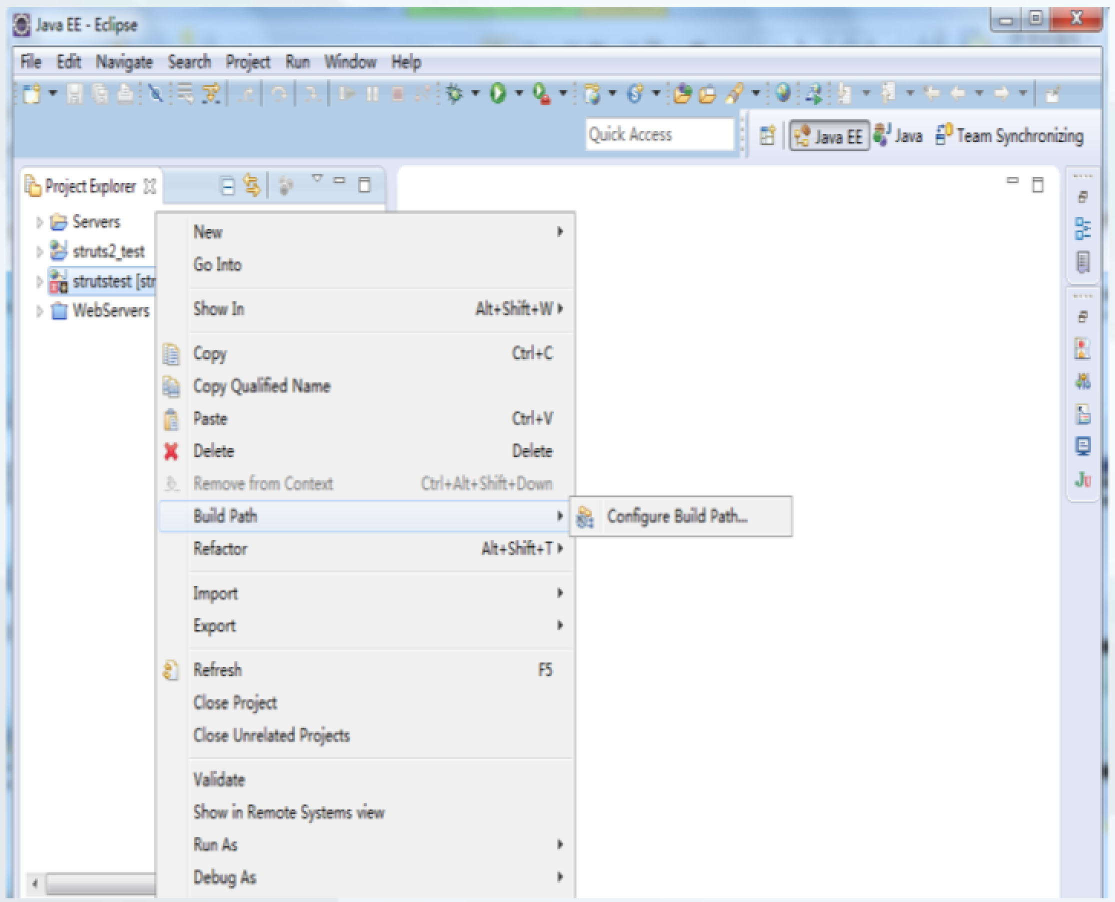
- **unit testing**: Looking for errors in a subsystem in isolation.
  - Generally a "subsystem" means a particular class or object.
  - The Java library **JUnit** helps us to easily perform unit testing.

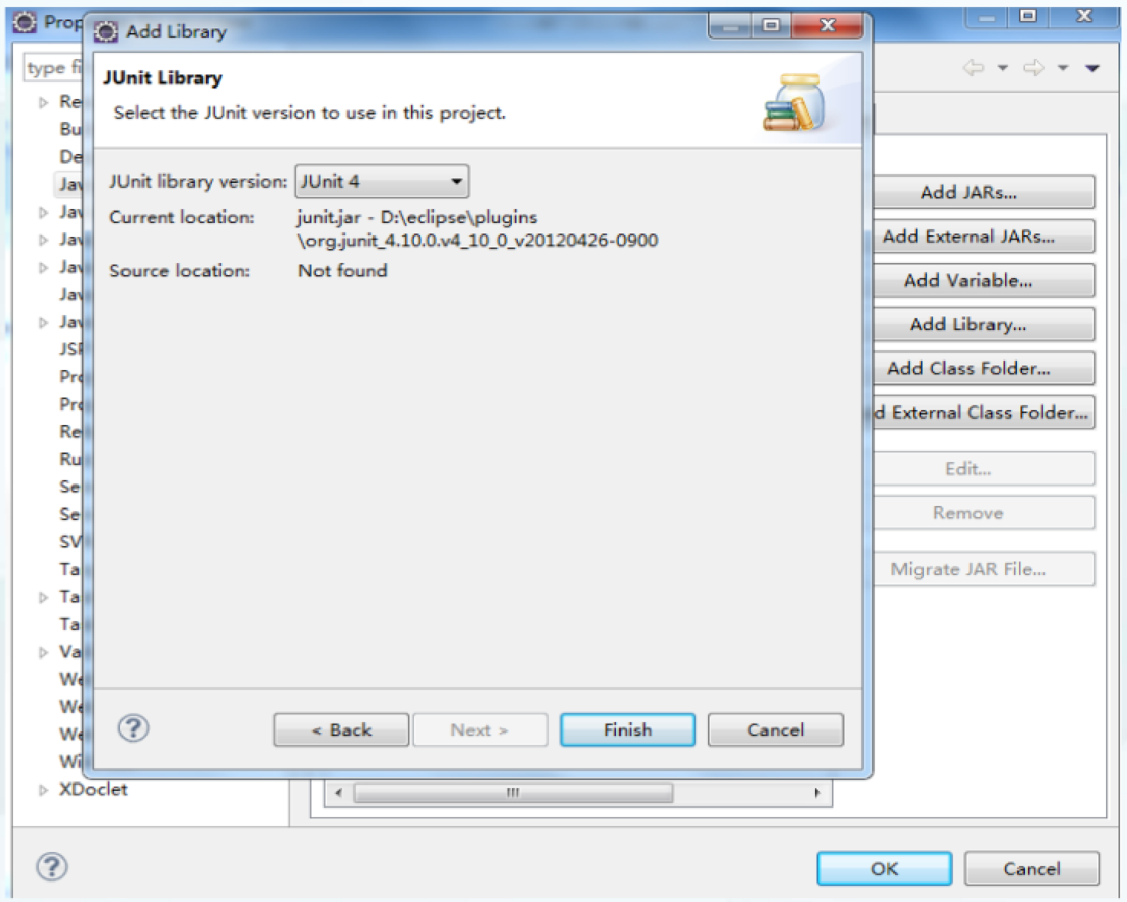
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.

- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish**





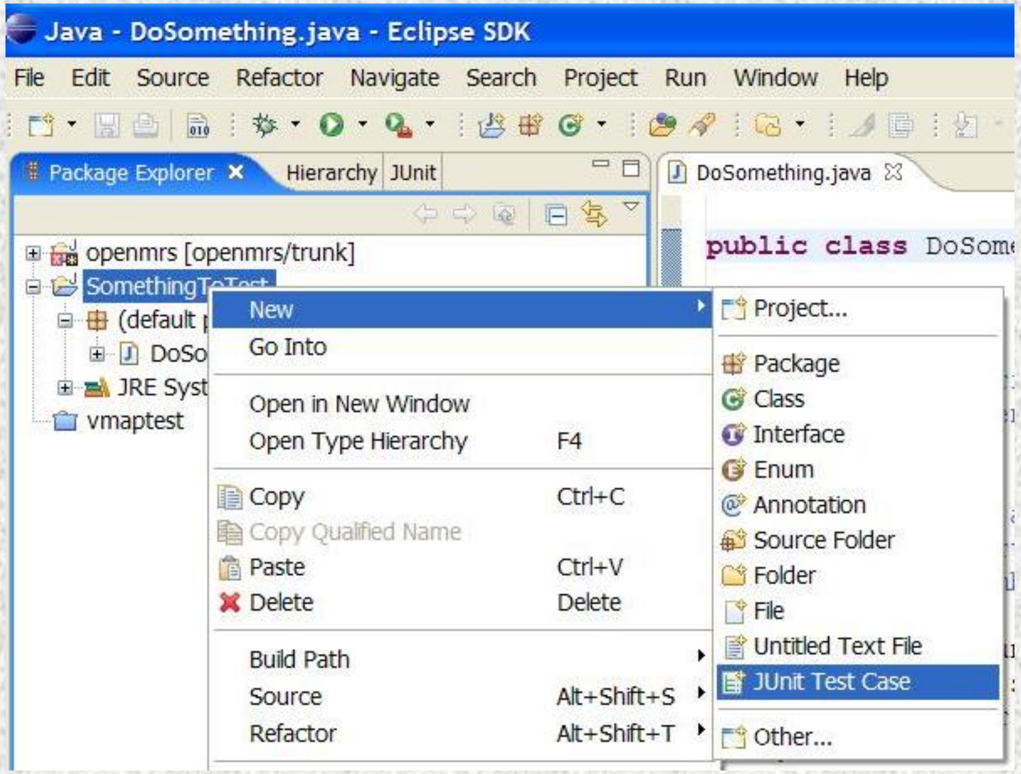
JUnit Library

Select the JUnit version to use in this project.

JUnit library version: JUnit 4
Current location: junit.jar - D:\eclipse\plugins\org.junit_4.10.0.v4_10_0_v20120426-0900
Source location: Not found

JUnit and Eclipse

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or click **File → New → JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



A JUnit test class

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

public class name {
    ...

    @Test
    public void name() {   // a test case method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

JUnit assertion methods

`assertTrue(`**`test`**`)`	fails if the boolean test is `false`
`assertFalse(`**`test`**`)`	fails if the boolean test is `true`
`assertEquals(`**`expected`**`, `**`actual`**`)`	fails if the values are not equal
`assertSame(`**`expected`**`, `**`actual`**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**`expected`**`, `**`actual`**`)`	fails if the values *are* the same (by ==)
`assertNull(`**`value`**`)`	fails if the given value is *not* `null`
`assertNotNull(`**`value`**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals(`**`"message"`**`, `**`expected`**`, `**`actual`**`)`

ArrayIntList JUnit test

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

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
```

JUnit annotations

- @BeforeClass – Run once before any of the test methods in the class, public static void
- @AfterClass – Run once after all the tests in the class have been run, public static void
- @Before – Run before @Test, public void
- @After – Run after @Test, public void
- @Test – This is the test method to run, public void

JUnit annotations

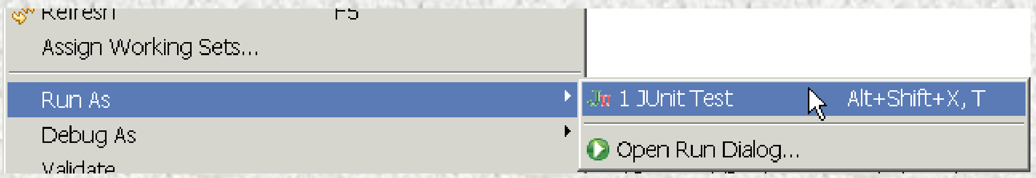
```
@BeforeClass - runOnceBeforeClass

@Before - runBeforeTestMethod
@Test - test_method_1
@After - runAfterTestMethod

@Before - runBeforeTestMethod
@Test - test_method_2
@After - runAfterTestMethod

@AfterClass - runOnceAfterClass
```

```java
public class BasicAnnotationTest {

    // Run once, e.g. Database connection, connection pool
    @BeforeClass
    public static void runOnceBeforeClass() {
        System.out.println("@BeforeClass - runOnceBeforeClass");
    }

    // Run once, e.g close connection, cleanup
    @AfterClass
    public static void runOnceAfterClass() {
        System.out.println("@AfterClass - runOnceAfterClass");
    }

    // Should rename to @BeforeTestMethod
    // e.g. Creating an similar object and share for all @Test
    @Before
    public void runBeforeTestMethod() {
        System.out.println("@Before - runBeforeTestMethod");
    }

    // Should rename to @AfterTestMethod
    @After
    public void runAfterTestMethod() {
        System.out.println("@After - runAfterTestMethod");
    }

    @Test
    public void test_method_1() {
        System.out.println("@Test - test_method_1");
    }

    @Test
    public void test_method_2() {
        System.out.println("@Test - test_method_2");
    }

}
```

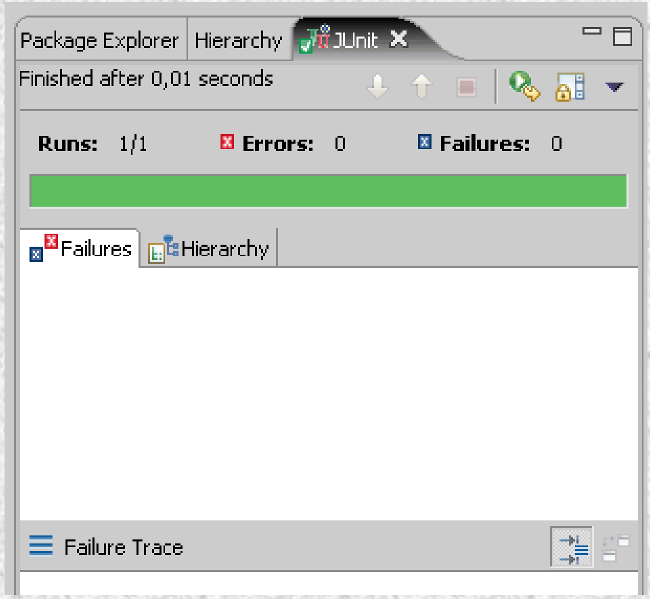
Running a test

- Right click it in the Eclipse Package Explorer at left; choose:

  Run As → JUnit Test



- The JUnit bar will show **green** if all tests pass, **red** if any fail.
- The Failure Trace shows which tests failed, if any, and why.



Finished after 0,01 seconds

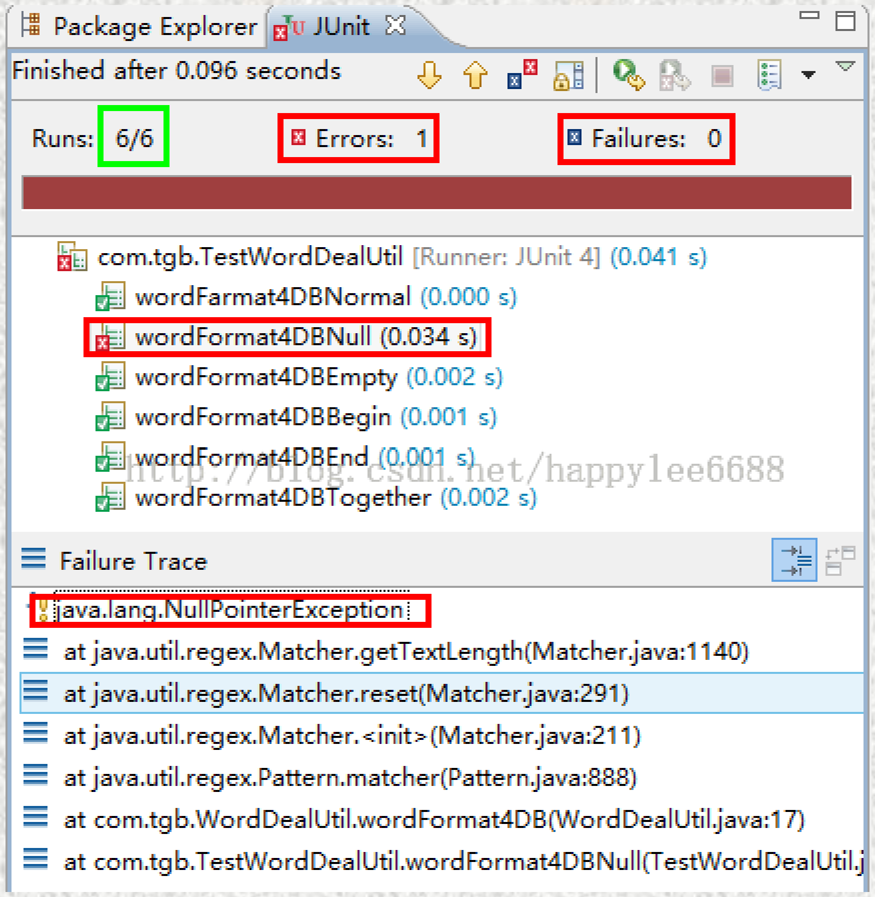
Runs: 1/1 Errors: 0 Failures: 0

Running a test

- Right click it in the Eclipse Package Explorer at left; choose:

  Run As → JUnit Test

- The JUnit bar will show **green** if all tests pass, **red** if any fail.
- The Failure Trace shows which tests failed, if any, and why.



```
Finished after 0.096 seconds
Runs: 6/6        Errors: 1        Failures: 0

com.tgb.TestWordDealUtil [Runner: JUnit 4] (0.041 s)
    wordFarmat4DBNormal (0.000 s)
    wordFormat4DBNull (0.034 s)
    wordFormat4DBEmpty (0.002 s)
    wordFormat4DBBegin (0.001 s)
    wordFormat4DBEnd (0.001 s)
    wordFormat4DBTogether (0.002 s)

Failure Trace
java.lang.NullPointerException
at java.util.regex.Matcher.getTextLength(Matcher.java:1140)
at java.util.regex.Matcher.reset(Matcher.java:291)
at java.util.regex.Matcher.<init>(Matcher.java:211)
at java.util.regex.Pattern.matcher(Pattern.java:888)
at com.tgb.WordDealUtil.wordFormat4DB(WordDealUtil.java:17)
at com.tgb.TestWordDealUtil.wordFormat4DBNull(TestWordDealUtil.j
```

JUnit exercise

Given a `Date` class with the following methods:

```
• public Date(int year, int month, int day)
• public Date()                           // today
• public int getDay(), getMonth(), getYear()
• public void addDays(int days)    // advances by days
• public int daysInMonth()
• public String dayOfWeek()        // e.g. "Sunday"
• public boolean equals(Object o)
• public boolean isLeapYear()
• public void nextDay()            // advances by 1 day
• public String toString()
```

- Come up with unit tests to check the following:
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

What's wrong with this?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
}
```

`assertTrue(`**test**`)`
`assertFalse(`**test**`)`
`assertEquals(`**expected**`, `**actual**`)`
`assertSame(`**expected**`, `**actual**`)`
`assertNotSame(`**expected**`, `**actual**`)`
`assertNull(`**value**`)`
`assertNotNull(`**value**`)`
`fail()`

Well-structured assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());   // expected
        assertEquals(2, d.getMonth());     // value should
        assertEquals(19, d.getDay());      // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

Expected answer objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);   // use an expected answer
    }                                // object to minimize tests

                                     // (Date must have toString
    @Test                            //  and equals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

Naming test cases

```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

What's wrong with this?

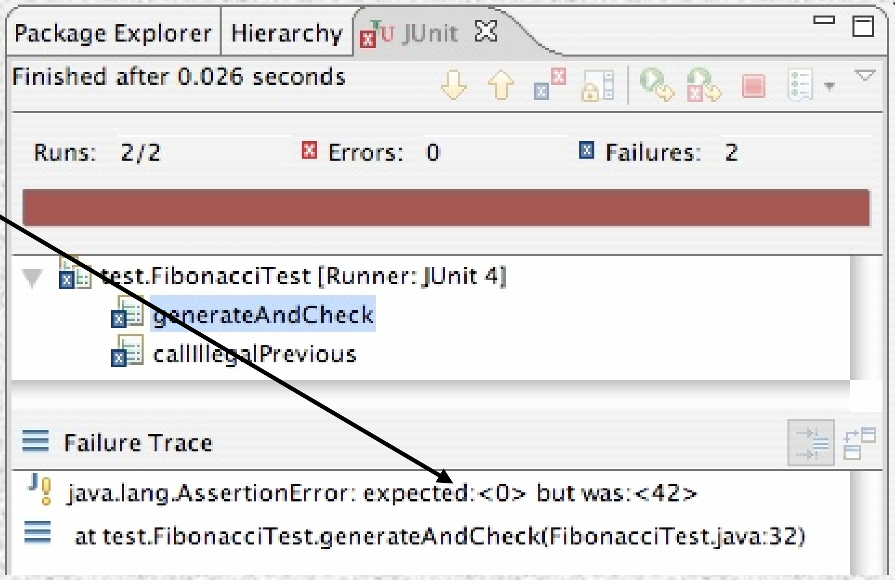
```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

Unnecessary message

Good assertion messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

```
// JUnit will already show
// the expected and actual
// values in its output;
//
// don't need to repeat them
// in the assertion message
```



Finished after 0.026 seconds

```
java.lang.AssertionError: expected:<0> but was:<42>
at test.FibonacciTest.generateAndCheck(FibonacciTest.java:32)
```

Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Pervasive timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, d);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // almost every test should have a timeout so it can't
    // lead to an infinite loop; good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Testing for exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it **does** throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);    // should throw the exeception
}
```

Setup and teardown

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- methods to run before/after each test case method is called

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- methods to run once before/after the entire test class runs

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- Test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

Trustworthy tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.

- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.

- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.

Squashing redundancy

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_withinSameMonth_1() {
        addHelper(2050, 2, 15, +4, 2050, 2, 19);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_wrapToNextMonth_2() {
        addHelper(2050, 2, 15, +14, 2050, 3, 1);
    }

    // use lots of helpers to make actual tests extremely short
    private void addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date act = new Date(y, m, d);
        actual.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", exp, act);
    }

    // can also use "parameterized tests" in some frameworks
    ...
```

Flexible helpers

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
        addhelper(d, +32, 2050, 4, 2);
        addhelper(d, +98, 2050, 7, 9);
    }

    // Helpers can box you in; hard to test many calls/combine.
    // Create variations that allow better flexibility
    private Date addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date date = new Date(y, m, d);
        addHelper(date, add, y2, m2, d2);
        return d;
    }

    private void addHelper(Date date, int add,
                           int y2, int m2, int d2) {
        date.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("date after +" + add + " days", expect, d);
    }
    ...
```

overload

Test-driven development

- Unit tests can be written after, during, or even *before* coding.
    - **test-driven development**: Write tests, *then* write code to pass them.

- Imagine that we'd like to add a method `subtractWeeks` to our `Date` class, that shifts this `Date` backward in time by the given number of weeks.

- Write code to test this method *before* it has been written.
    - Then once we do implement the method, we'll know if it works.

Test case "smells"

- Tests should be self-contained and not care about each other.

- "**Smells**" (bad things to avoid) in tests:
  - *Constrained test order*: Test A must run before Test B.
    (usually a misguided attempt to test order/flow)
  - *Tests call each other* : Test A calls Test B's method
    (calling a shared helper is OK, though)
  - *Mutable shared state* : Tests A/B both use a shared object.
    (If A breaks it, what happens to B?)