Unit Testing with JUnit

Refresher

Richard Paige

Introduction

- Unit testing is something you have likely done in one of your courses, in your own projects, etc.

- You may have used unit testing tools (like *Unit) in the past.

- This set of notes is a refresher on one of these tools – JUnit for Java – with some examples and exercises.
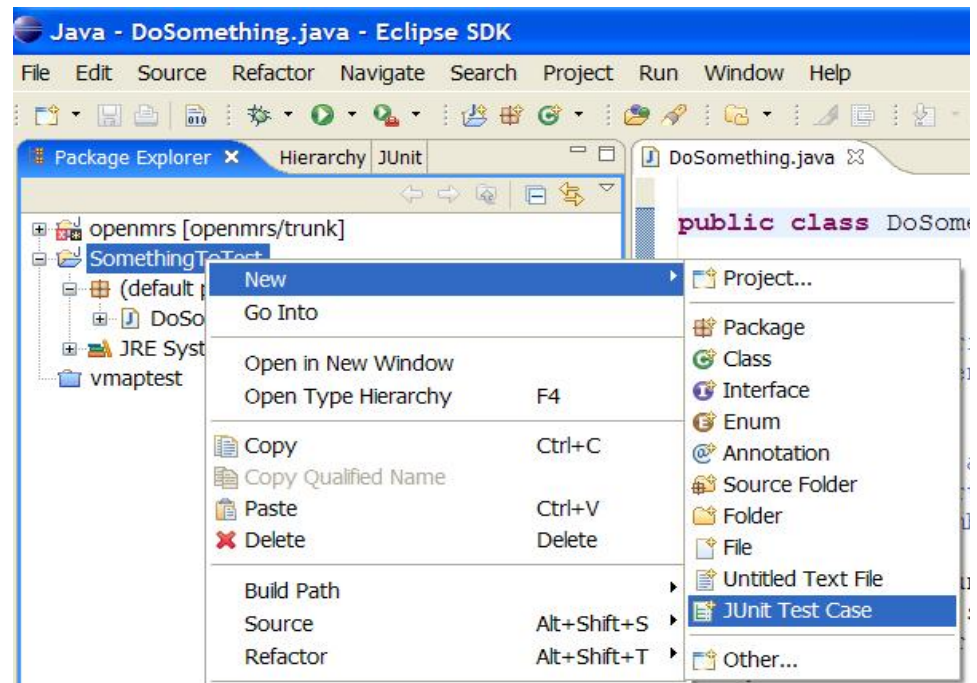
Unit testing

- **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.
  - Test case methods look 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

- Current JDT has JUnit available by default.
- You can manually add it if you must, e.g., via
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 5 → Finish**

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or select **File → New → JUnit Test Case**

  - Eclipse can create stubs of method tests for you.

A JUnit test class

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

public class nameTest {
    ...
 @Test
 public void test1name() {   // 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**)

  ➢ Why is there no pass method?

ArrayIntList JUnit test

```java
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());
    }
```

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.
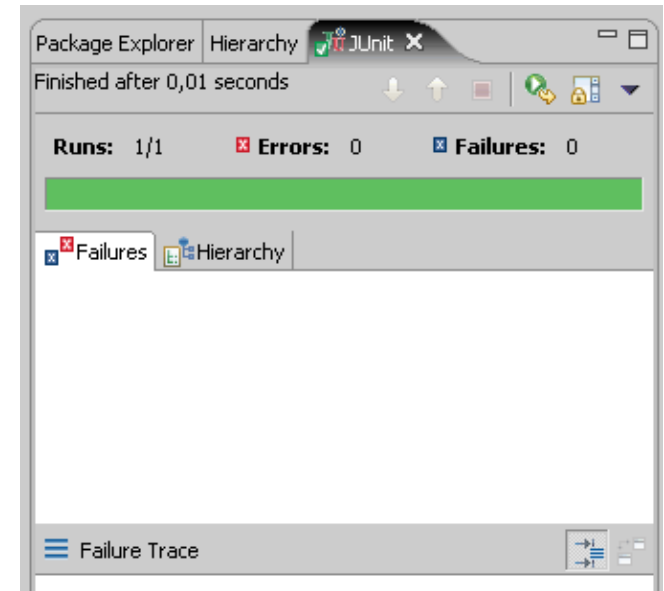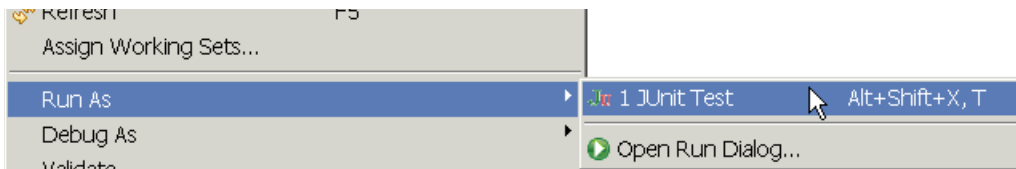
JUnit example

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 no `Date` object can ever get into an invalid state.

  ➢ 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?

```java
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);
    }
}
```

Well-structured assertions

```java
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

```java
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

```java
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?

```java
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);
    }
    ...
}
```

Good assertion messages

```java
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
```
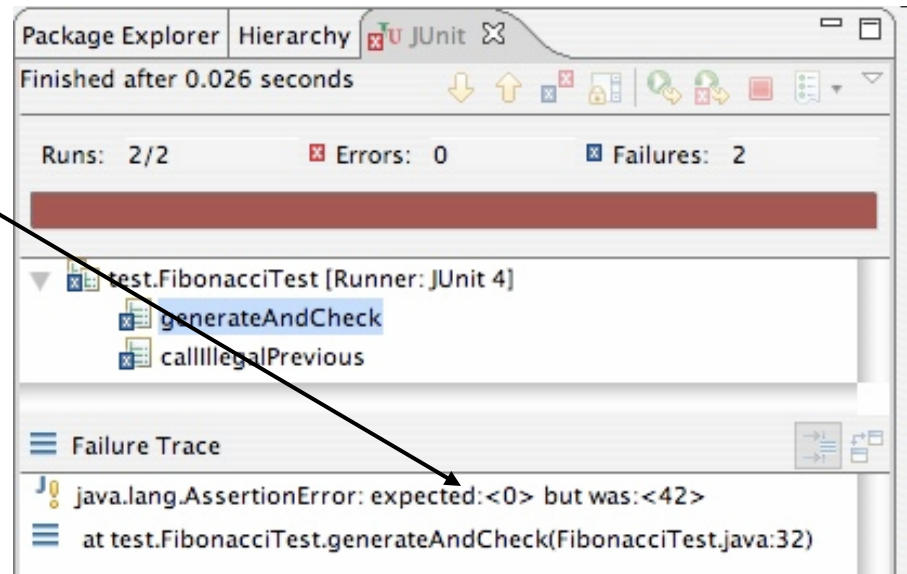
Tests with a timeout

```
@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

```java
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

```java
@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 conditions.

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

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

What's wrong with this?

```java
public class DateTest {
    // test every day of the year
    @Test(timeout = 10000)
    public void tortureTest() {
        Date date = new Date(2050, 1, 1);
        int month = 1;
        int day = 1;
        for (int i = 1; i < 365; i++) {
            date.addDays(1);
            if (day < DAYS_PER_MONTH[month]) {day++;}
            else                             {month++; day=1;}
            assertEquals(new Date(2050, month, day), date);
        }
    }

    private static final int[] DAYS_PER_MONTH = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    }; // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
}
```

JUnit example

Given our `Date` class seen previously:

- `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 no `Date` object can ever get into an invalid state.
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

Squashing redundancy

```java
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

```java
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(y1, m1, d1);
        addHelper(date, add, y2, m2, d2);
        return date;
    }

    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);
    }
```

Regression testing

- **regression**: When a feature that used to work, no longer works.
  - ➤ Likely to happen when code changes and grows over time.
  - ➤ A new feature/fix can cause a new bug or reintroduce an old bug.

- **regression testing**: Re-executing prior unit tests after a change.
  - ➤ Often done by scripts during automated testing.
  - ➤ Used to ensure that old fixed bugs are still fixed.
  - ➤ Gives your app a minimum level of working functionality.

- Many products have a set of mandatory check-in tests that must pass before code can be added to a source code repository.

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.

Tests and data structures

- Need to pass lots of arrays? Use array literals
  ```
  public void exampleMethod(int[] values) { ... }
  ...
  exampleMethod(new int[] {1, 2, 3, 4});
  exampleMethod(new int[] {5, 6, 7});
  ```

- Need a quick `ArrayList`? Try `Arrays.asList`
  ```
  List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
  ```

- Need a quick set, queue, etc.? Many collections can take a list
  ```
  Set<Integer> list = new HashSet<Integer>(
                        Arrays.asList(7, 4, -2, 9));
  ```

What's wrong with this?

```java
public class DateTest {
    // shared Date object to test with (saves memory!!)
    private static Date DATE;

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_sameMonth() {
        DATE = new Date(2050, 2, 15);        // first test;
        addhelper(DATE,  +4, 2050, 2, 19); // DATE = 2/15 here
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_nextMonthWrap() {  // second test;
        addhelper(DATE, +10, 2050, 3, 1);  // DATE = 2/19 here
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls() {  // third test;
        addDays_sameMonth();                 // go back to 2/19;
        addhelper(DATE, +1, 2050, 2, 20);  // test two calls
        addhelper(DATE, +1, 2050, 2, 21);
    }
    ...
}
```

Test case "smells"

- Tests should be self-contained and not depend on 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?)

Test suites

- **test suite**: One class that runs many JUnit tests.
  - ➤ An easy way to run all of your application's tests at once.

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestCaseName.class,
    TestCaseName.class,
    ...
    TestCaseName.class,
})
public class name {}
```

Test suite example

```java
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

JUnit summary

- Tests need *failure atomicity* (ability to know exactly what failed).
  - ➤ Each test should have a clear, long, descriptive name.
  - ➤ Assertions should always have clear messages to know what failed.
  - ➤ Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.

- Always use a `timeout` parameter to every test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@Before` to reduce redundancy between tests.