JUnit

ENGI 5895

Adapted from slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia

http://www.cs.washington.edu/331/

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.

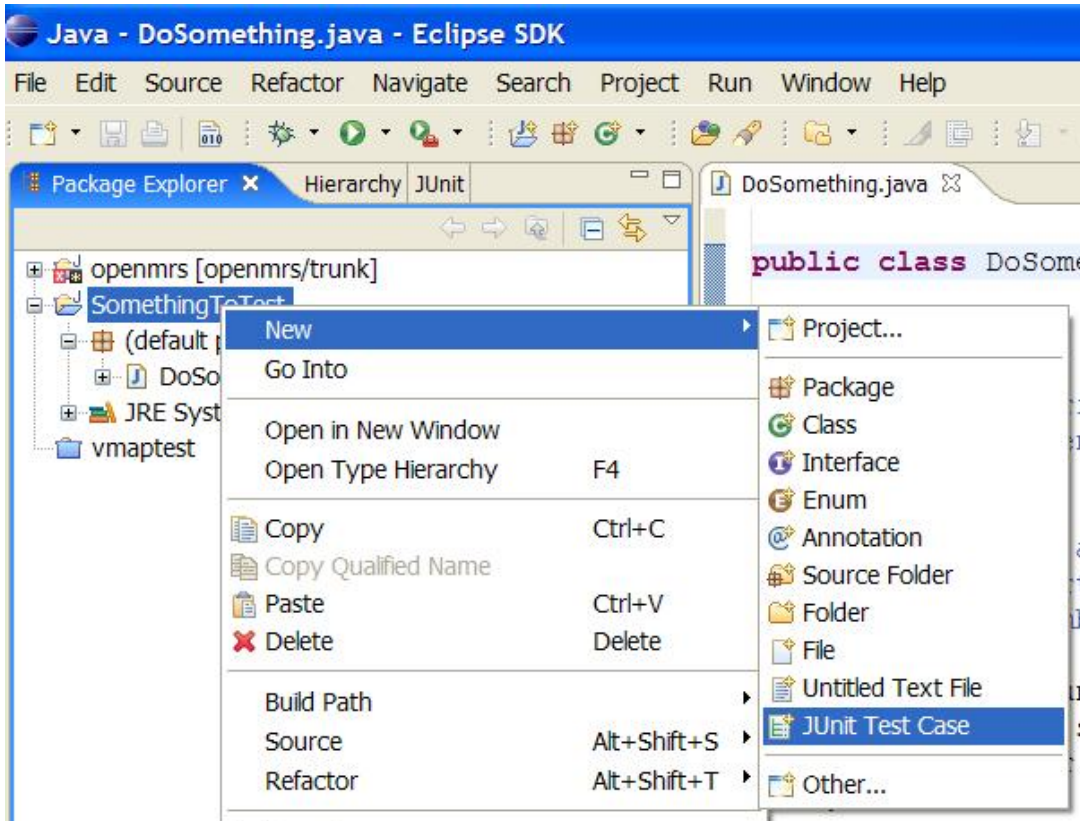
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.

- 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**

- 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());
    }
    ...
```

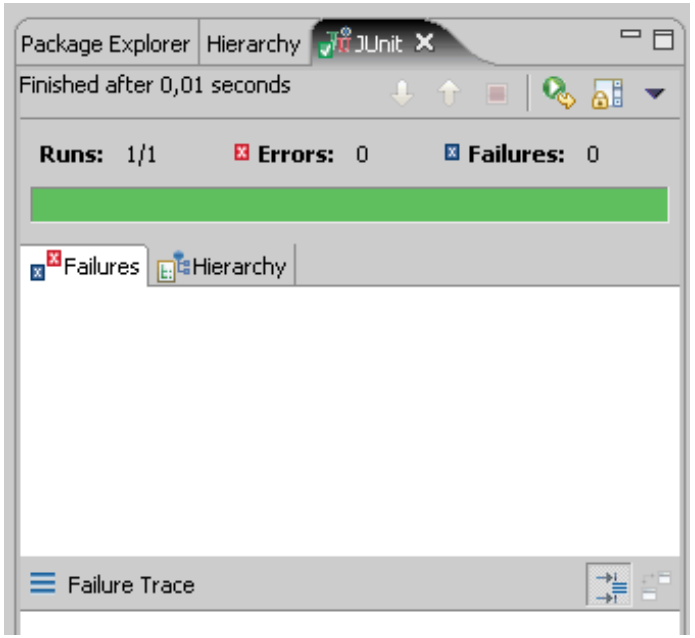
Running a test

- 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 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()`

- Test the `addDays` method

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);
    }
}
```

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() {
        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() {
        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
}
```

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

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 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

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.
- Test for expected errors / exceptions.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@Before` to reduce redundancy between tests.