Unit Testing part 1

CSCI 2300

Time it takes to test

Manual (to run each test case)	Programmatic (to write each test case)
31.2 seconds	34.6 seconds
48 seconds	50 seconds
20 seconds	26 seconds
50 seconds	36 seconds
12 seconds	36 seconds

Why would we want to add programmatic tests?

Because we can reuse them over and over again!

Bugs and testing

- **Software reliability**: Probability that a software system will not cause failure under specified conditions.
  - Measured by uptime, MTTF (mean time till failure), crash data.
- **Bugs** are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.
- **Testing**: A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation).

Difficulties of testing

- Perception by some developers and managers:
  - Testing is seen as a novice's job.
  - Assigned to the least experienced team members.
  - Done as an afterthought (if at all).
    - "My code is good; it won't have bugs. I don't need to test it."
    - "I'll just find the bugs by running the client program."
- Limitations of what testing can show you:
  - It is impossible to completely test a system.
  - Testing does not always directly reveal the actual bugs in the code.
  - Testing does not prove the absence of errors in software.

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.

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.

Example

```
public class Day
{
   public Day(int year, int month, int day);
   public int getYear();
   public int getMonth();
   public int getDay();
   public Day addDays(int n);
   public int daysFrom(Day other);
}
```

```
public class DayTest
{
   @Test
   public void constructorTest1()
   {
      Day d = new Day(2019, 4, 3);
      assertEquals(2019, d.getYear());
      assertEquals(4, d.getMonth());
      assertEquals(3, d.getDay());
   }
}
```


```
public class Day
{
   public Day(int year, int month, int day);
   public int getYear();
   public int getMonth();
   public int getDay();
   public Day addDays(int n);
   public int daysFrom(Day other);
}
```


```
1. public class DayTest
2. {
3.    @Test
4.    public void daysFromTest()
5.    {
6.       Day d1 = new Day(2019, 4, 3);
7.       d1.addDays(5);
8.       Day d2 = new Day(2019, 4, 3);
9.       int daysFrom = d1.daysFrom(d2);
10.      assertEquals(5, daysFrom);
11.   }
12. }
```

Suppose daysFromTest fails on line 10. This means there is a bug in:

A. daysFrom() method
B. addDays() method
C. Day constructor
D. All of the above
E. Can't tell for sure

Unit Testing Rule #1

Each unit test must focus on testing one behavior.

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

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

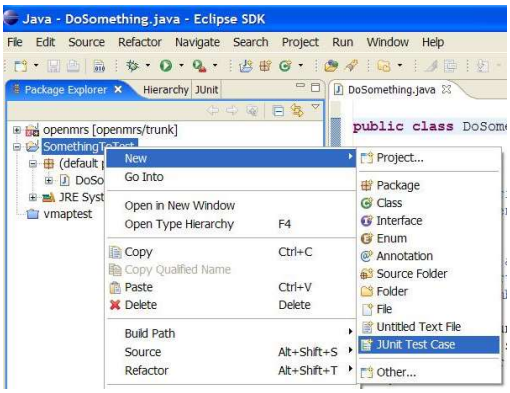
JUnit on command line on hopper.slu.edu

- javac -cp <path to junit.jar>:. MyTest.java
- Example:
  javac -cp /usr/share/java/junit.jar:. DayTest.java
- You can set CLASSPATH variable (see example in `unitTesting/set_classpath.sh`)
  - In ~/.bashrc add:
    `CLASSPATH=$CLASSPATH:/usr/share/java/junit.jar:.`
  - After modifying bashrc, run:
    (only the first time) `source ~/.bashrc`
- Run your test:
  java -cp $CLASSPATH org.junit.runner.JUnitCore DayTest
- You can use an 'alias' to simplify your life:
  - In ~/.bashrc add:
    `alias junit="java -cp $CLASSPATH org.junit.runner.JUnitCore"`
  - Run: `source ~/.bashrc` (only once after modifying bashrc)
  - Run: `junit DayTest`

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.

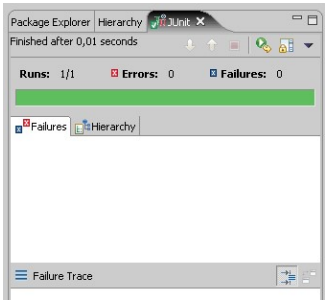


Running a test

- 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.



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
```

Unit testing rule #2

Assertions should use descripting messages, explaining what is being checked.

assertEquals()

- assertEquals() on primitive types compares the values
  Example:
```
int days = …
assertEquals(5, days);
```
- assertEquals() on class types calls `equals()` method on the 'expected' object
  Example:
```
Day d1 = …
Day d2 = …
assertEquals(d1, d2)
```
➡ Calls `d1.equals(d2)`

`Object`
`+equals(Object): boolean` ➡ Compares references

Overriding `equals()` method

```
public class Day
{
   protected int day;
   protected int month;
   protected int year;
   @Override
   boolean equals(Object o)
   {
       boolean result = false;
       if (this.day == o.day &&
           this.month == o.month &&
           this.year == o.year)
       {
          result = true;
       }
       return result;
   }
}
```

There is a problem with this code because:

A. Day cannot Override any methods because it does not have a parent class.
B. `Object o` does not have `day, month, year` instance variables
C. `equals()` method takes `Object` as an argument, it should be `Day` instead
D. All of the above
E. There is no problem with this code.

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
}
```

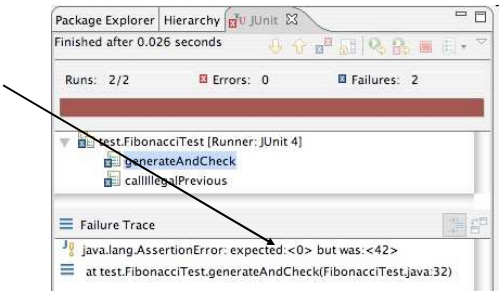
Unit testing rule #3

Use descriptive names for test case methods and expected/actual variables.

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
```



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;
}
```

Unit testing rule #4

Limit the execution time of each unit test.