Java Programming 2
Lecture #18

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
20 November 2019

Schedule for the rest of the semester

LECTURES/TUTORIALS

Week 9:

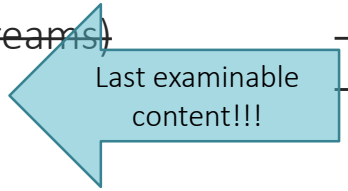
- ~~18 November: Lecture (Enum types, streams)~~
- 20 November: Lecture (JUnit, Javadoc)
- 22 November: No tutorial

Week 10:

- 25 November: Quiz (1% credit available)
- 27 November: Revision lecture
- 29 November: Tutorial: lab exam prep

Week 11:

- 2 December: No lecture (lab exam)
- 4 November, 6 November: going over past exam problems



LABS

Lab 8

- ~~15 November: Lab 8 distributed~~
- ~~18/19 November: work on Lab 8 in lab~~
- 21 November: Lab 8 due

Lab exam

- 20 November: Lab exam practice problem distributed
- 25/26 November: work on practice problem with tutor help
  - *Highly recommended especially if you have been using your own computer, ESPECIALLY if you have not been using Eclipse*
- 2/3 December: Lab exam

Outline

Software testing overview

Unit testing

Writing unit tests with JUnit

Javadoc comments

Software testing levels

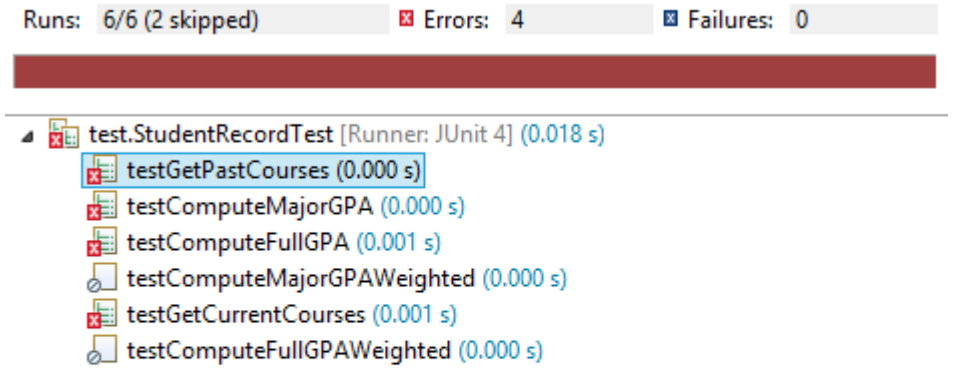
Unit testing
- Verify the functionality of a **specific section of code**
- Tests usually written by developers (“white-box”)

Integration testing
- Verify the **interfaces between components**
- Exposes defects in the interfaces and interactions between modules

System testing (end-to-end testing)
- Tests a **complete integrated system** to ensure that it meets requirements

Test-driven development a.k.a. “Red-green refactoring”

1. Write a new test
2. Run all tests (newly added test should fail)
3. Write the implementation code to make the test pass (KISS – Keep It Simple Stupid)
4. Run all tests (tests should now pass)
5. Refactor
6. Repeat



https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/

Benefits of unit testing

Find problems early in the development cycle

Support refactoring and other code changes

Potentially simplify integration testing

Provide “living documentation” of the system

Unit tests can provide a design document for the class

Limitations of unit testing

Cannot catch every error in the program – cannot cover every possible execution path

Cannot catch integration or system-level errors

Complexity of setting up realistic tests – creating initial conditions

Tests could be buggy themselves

Cannot easily test problems that use randomness or multiple threads

Writing a unit test

A good unit test is

- **Atomic:** it tests exactly one piece of code, does not depend on any other tests, and can be run repeatedly and/or concurrently
- **Trustworthy:** it should run every time on every machine
- **Maintainable:** test code is code too – avoid repetition, magic numbers, etc
- **Readable:** code should make sense; names should describe what they are testing

https://philippetruche.wordpress.com/2011/03/22/unit-testing-101-fundamentals-of-unit-testing/

What does a unit test actually do?

1. Set up a predictable context
2. Run a method and check that the result meets a specification

   Return value correct?

   Expected exception thrown?

   Correct side effects detected?


JUnit

Overview of JUnit

"A simple framework to write repeatable tests." (http://junit.org/junit4/index.html)

Widely used – over 30% of Java projects on github.org make use of JUnit tests (http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/)

History

- Started in 1998 as SUnit (for Smalltalk)
- Ported to Java, and eventually other OO languages (Ruby, etc.) – collectively known as "xUnit"

Component architecture of xUnit

Test runner

A program that runs the tests and reports the results

Test case

A single unit test case

Test fixtures

A set of preconditions (i.e., state) needed to run a test

Test suite

A set of tests that all share the same fixture – ordering should not matter

A sample JUnit test

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class MyTests {
        @Test
        public void multiplicationOfZeroIntegersShouldReturnZero() {
                MyClass tester = new MyClass(); // MyClass is tested

                // assert statements
                assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
                assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
                assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));

        }
}
```

http://www.vogella.com/tutorials/JUnit/article.html

Naming conventions

Widely used:

- Add suffix “Test” to the name of the test classes
- Create them in a new package “test”

Name should explain what the test does

- (If you use good names, you don’t need to read the code)
- One possible convention: use “should” in the method name
  - *E.g., “addShouldThrowExceptionOnNull”)*

Defining tests in JUnit: use annotations

@Test: identifies a method as a test method

@Test (expected = Exception.class) – fails unless the given exception is thrown

@Test (timeout = 100) – fails if it takes more than 100 milliseconds

@BeforeEach: executed before each test is run (sets up the fixture(s))

@AfterEach: executed after each test is completed (deletes fixture(s))

@BeforeAll: executed once before all tests in the file are run (e.g., connect to database)

@AfterAll: executed once after all tests in the file are run (e.g., disconnect)

@Ignore: indicates that the given test should be ignored

@Ignore("Reason") also gives an explanation – good practice to include

Writing the body of a test – use Assert

Used to check the actual result against an expected result

Optional (but recommended): specify the string to display if they do not match

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

import org.junit.Test;

public class MyTests {

        @Test
        public void multiplicationOfZeroIntegersShouldReturnZero() {
                MyClass tester = new MyClass(); // MyClass is tested

                // assert statements
                assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
                assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
                assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
        }
}
```

http://www.vogella.com/tutorials/JUnit/article.html

Aside: “import static”?

Allows unqualified access to **all static members** in the imported class

E.g., instead of

```
int m = Math.max (a, b);
```

You could write

```
import static Math.max; // (or import static Math.*)
// …
int m = max(a, b);
```

“[W]hen should you use static import? **Very sparingly!**”
(https://docs.oracle.com/javase/1.5.0/docs/guide/language/static-import.html)

Assertions (all in org.junit.Assert class)

Assertion	Notes
fail([message])	Causes the test to fail – e.g., if it's not written yet
assertTrue([message], condition)	Just checks true/false on condition (may be complex)
assertFalse([message], condition)	
assertEquals([message], expected, actual)	Uses .equals()
assertNotEquals([message], expected, actual)	
assertEquals([message], expected, actual, delta)	Compares floating-point numbers with a tolerance
assertNotEquals([message], expected, actual, delta)	
assertArrayEquals([message], expected, actual)	Compares individual array values with .equals()
assertSame([message], expected, actual)	Uses ==
assertNotSame([message], expected, actual)	

How many assertions per test?

It depends …

Each test case should only potentially fail for **one reason**

But that reason could potentially involve multiple assertions

```
@Test
public void testResultShouldBeInRange() {
      int result = obj.getResult();
      assertTrue("Result too small", result > 0);
      assertTrue("Result too big", value < 100);
}
```

Advanced JUnit – using matchers

One more method in Assert: `assertThat(message, Matcher<T>)`

Sample uses :

```
assertThat(actual, is(equalTo(expected)));
assertThat(actual, is(not(equalTo(expected))));
assertThat(actual, containsString(expected));
assertThat(123, is("abc"));      //does not compile
assertThat("test", anyOf(is("test2"), containsString("ca")));
```

More information and examples at https://objectpartners.com/2013/09/18/the-benefits-of-using-assertthat-over-other-assert-methods-in-unit-tests/ (source of the above examples)

Javadoc comments

Javadoc tool

- Processes Java source files and generates HTML documentation
  - This is what generates the Java class documentation at https://docs.oracle.com/javase/8/docs/api/
- Reflects the structure of the source files
- Also includes any content in specially-formatted comments

Constructor Summary

Constructors

Constructor and Description
`String()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte)` **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.
`String(byte[] bytes, int offset, int length)` Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] bytes, int offset, int length, Charset charset)` Constructs a new `String` by decoding the specified subarray of bytes using the specified **charset**.

Javadoc comments

- Surrounded by **/** … */**
  - Blue rather than green in Eclipse
- Refer to the class member that comes directly below them
  - Classes
  - Interfaces
  - Constructors
  - Fields
  - Methods
- Can contain HTML

```
/** Class Description of MyClass
*/
public class MyClass {
	/** Field Description of
myIntField */
	public int myIntField;
	/** Constructor Description
of MyClass() */
	public MyClass() {
		// Do something ...
	}
}
```

https://students.cs.byu.edu/~cs240ta/fall2012/tutorials/javadoctutorial.html

Javadoc tags

Tags: keywords recognised by Javadoc to identify information

- @author *name*
- @version *version*
- @param *name description*
- @return *description*
- @throws *exception description*
- @see *other-class*

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* <p>
* Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
*
* @author  Zara Ali
* @version 1.0
*/

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");

    }
}
```

http://www.tutorialspoint.com/java/java_documentation.htm

Another example (CreditCard.java)

```
/**
 * Attempts to make a charge to the credit card.
 *
 * @param amount
 *            The amount to charge
 * @return true if the card still has enough room to make the charge, and
 *         false if not
 * @throws Exception
 *             if the amount to charge is invalid (i.e., not positive)
 */

public boolean charge(double amount) throws Exception {
    if (amount <= 0) {
        throw new Exception("Invalid charge amount: " + amount);
    }

    if ((amount + currentBalance) <= creditLimit) {
        currentBalance += amount;
        return true;
    } else {
        return false;
    }

}
```

Generating Javadoc comments in Eclipse

1. Click on element you want to document (class name, method name, etc.)
2. Press *alt-shift-J*
3. Edit the resulting auto-generated Javadoc to include details of the actual method/class/field/etc

Also, notice that when you click on an element in the Eclipse editor, you see some documentation in the “Javadoc” tag at the bottom

Is Javadoc required?

No, not strictly …

… but it is recommended to include it at least on classes/interfaces/methods

See, e.g., Monster.java sample solutions for more examples

Schedule for the rest of the semester

LECTURES/TUTORIALS

Week 9:

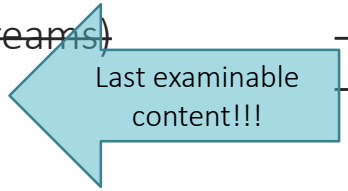
- ~~18 November: Lecture (Enum types, streams)~~
- 20 November: Lecture (JUnit, Javadoc)
- 22 November: No tutorial

Week 10:

- 25 November: Quiz (1% credit available)
- 27 November: Revision lecture
- 29 November: Tutorial: lab exam prep

Week 11:

- 2 December: No lecture (lab exam)
- 4 November, 6 November: going over past exam problems



LABS

Lab 8

- ~~15 November: Lab 8 distributed~~
- ~~18/19 November: work on Lab 8 in lab~~
- 21 November: Lab 8 due

Lab exam

- 20 November: Lab exam practice problem distributed
- 25/26 November: work on practice problem with tutor help
  - *Highly recommended especially if you have been using your own computer, ESPECIALLY if you have not been using Eclipse*
- 2/3 December: Lab exam

Next time

Friday: no tutorial

Monday: quiz (1% bonus credit!)

Wednesday: revision lecture