

# MSc/ICY Software Workshop

## Testing (Revisited)

### Functions

### Interfaces

Manfred Kerber [www.cs.bham.ac.uk/~mmk](http://www.cs.bham.ac.uk/~mmk)

26 October 2016

(partly based on material by Christoph Lange and Chris Bowers)

## How to test?

- All the code should be covered by tests (e.g. both branches of an if-then-else).
- You should test sufficiently many *typical* cases.
- You should test border cases (e.g. for an array `a` whether `a[0]` and `a[a.length-1]` are properly initialized/changed).
- You should write your code so that it is testable as far as possible (e.g., do not write a sophisticated print method since that cannot be tested by JUnit), but write a sophisticated `toString` method (since it can be tested by JUnit).

## A quote by E. W. Dijkstra (2000)

*A programmer has to be able to demonstrate that his program has the required properties. If this comes as an afterthought, it is all but certain that he won't be able to meet this obligation only if he allows this obligation to influence his design, there is hope that he can meet it. Pure a posteriori verification denies you that wholesome influence and is therefore putting the cart before the horse.*

# Test-Driven Development

- Program by intent

- 1 Start by defining a set of test cases

- Pay attention to border cases!

- 2 Write code that passes the tests

- ```
public class TestWordStemmer {  
    public void testStemmer() {  
        WordStemmer stemmer = new WordStemmer ();  
        assert stemmer.stem("helping") == "help";  
        assert stemmer.stem("hungrily") == "hungry";  
        assert stemmer.stem("friendliness") == "friend";  
        assert stemmer.stem("play") == "play";  
        assert stemmer.stem("playing") == "play";  
        assert stemmer.stem("player") == "play";  
        // ...  
    }  
}
```

# Test-Driven Development (Cont'd)

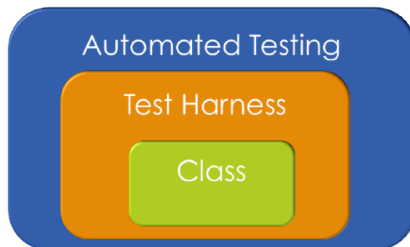
- Use tests as a template to create code
- 

```
public class WordStemmer {  
    public String stem(String word) {  
        String stemmed_word;  
        // do stemming  
        return stemmed_word;  
    }  
}
```

Find the right pattern to make defining test cases easier.

# Unit testing

- **Unit** small functional part of application that can be **tested independently**
- A unit could be an **individual class or method**
- A **set of test cases** are constructed to form a **test harness**



# JUnit testing with Eclipse

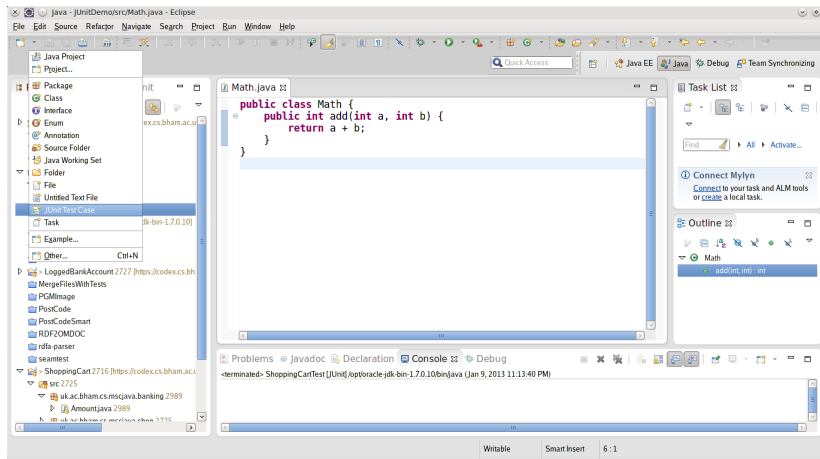
- JUnit: unit testing library for Java
- Integrated into Eclipse/NetBeans/XCode/... IDEs
- Separates source code from testing code
- IDEs provide lots of nifty inspection tools
- Well documented

## Outline:

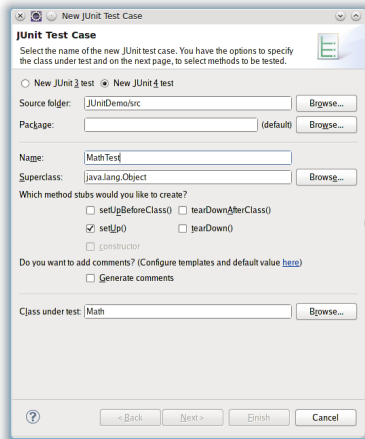
- ① creating JUnit test
- ② writing test cases
- ③ coding by intent
  - ① getting it wrong  $\Rightarrow$  test fails
  - ② getting it right  $\Rightarrow$  test passes



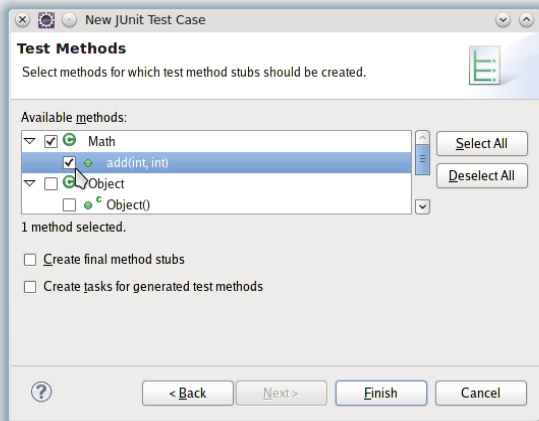
# Eclipse example: Creating JUnit test



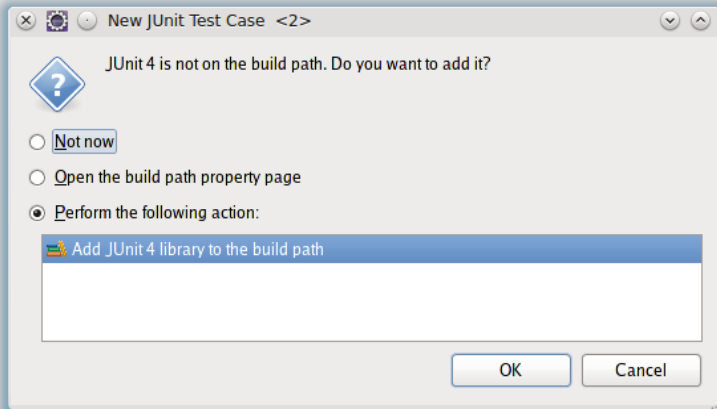
# Eclipse example (Cont'd): Creating JUnit test



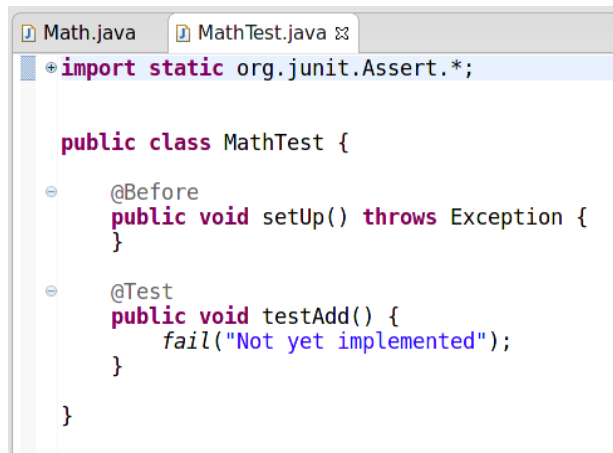
# Eclipse example (Cont'd): Creating JUnit test



# Eclipse example (Cont'd): Creating JUnit test

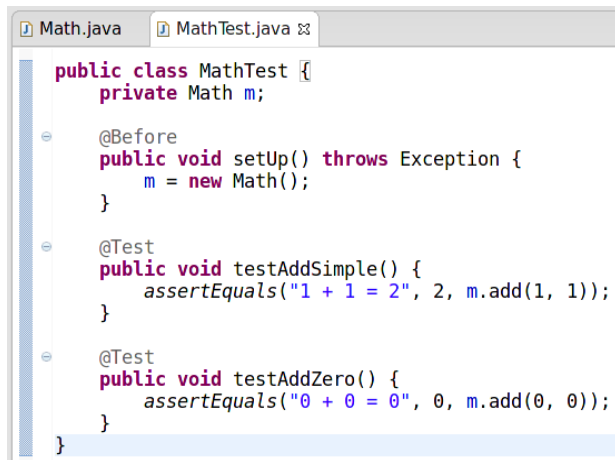


# Eclipse example (Cont'd): Creating JUnit test



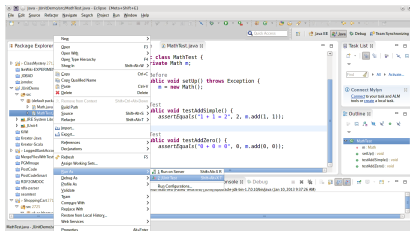
```
Math.java  MathTest.java ✖  
+ import static org.junit.Assert.*;  
  
public class MathTest {  
    @Before  
    public void setUp() throws Exception {  
    }  
  
    @Test  
    public void testAdd() {  
        fail("Not yet implemented");  
    }  
}
```

# Eclipse example (Cont'd): Writing test cases

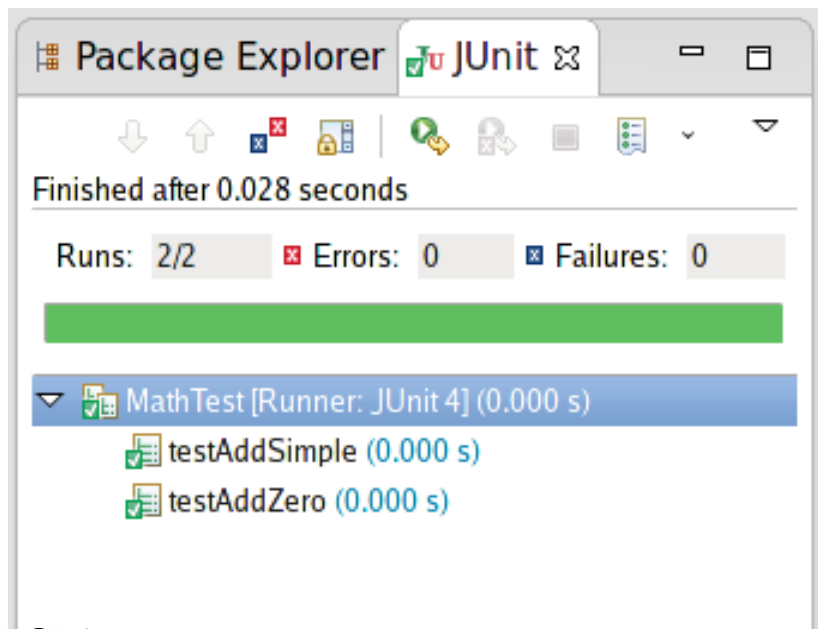


```
Math.java  MathTest.java ⌕  
  
public class MathTest {  
    private Math m;  
  
    @Before  
    public void setUp() throws Exception {  
        m = new Math();  
    }  
  
    @Test  
    public void testAddSimple() {  
        assertEquals("1 + 1 = 2", 2, m.add(1, 1));  
    }  
  
    @Test  
    public void testAddZero() {  
        assertEquals("0 + 0 = 0", 0, m.add(0, 0));  
    }  
}
```

# Eclipse example (Cont'd): Running a test



## Eclipse example (Cont'd): Running a test

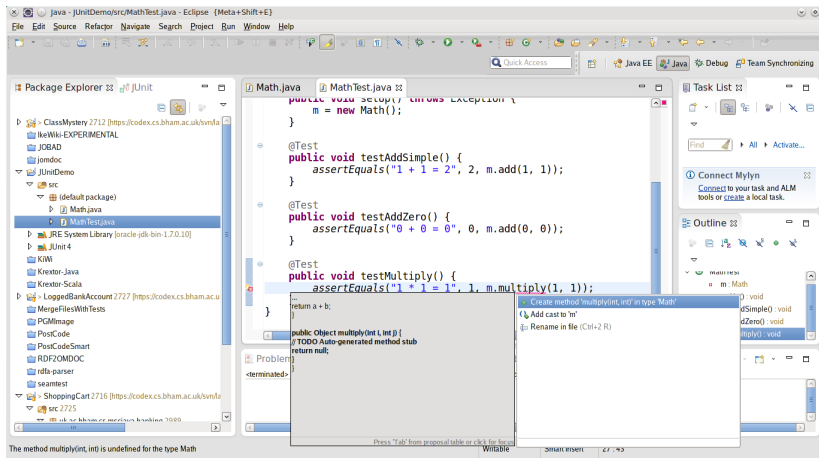





# Eclipse example (Cont'd): Test-driven development

```
@Test
public void testMultiply() {
    assertEquals("1 * 1 = 1", 1, m.multiply(1, 1));
}
```

# Eclipse example (Cont'd): Test-driven development



# Eclipse example (Cont'd): Test-driven development

A small icon of an Eclipse IDE editor window, showing a blue vertical bar on the left and a minus sign in a circle on the right.

```
public int multiply(int a, int b) {  
    return a - b;  
}
```

# Eclipse example (Cont'd): Test-driven development

The screenshot shows the Eclipse IDE's Package Explorer and JUnit test runner. The Package Explorer is titled 'JUnit' and shows a tree view of test results. The test runner indicates that the tests finished after 0.023 seconds. The summary shows 3 runs, 0 errors, and 1 failure. The test runner is titled 'MathTest [Runner: JUnit 4] (0.000 s)'. The test runner shows three tests: 'testMultiply (0.000 s)' which failed, 'testAddSimple (0.000 s)' which passed, and 'testAddZero (0.000 s)' which passed. The failure trace shows an 'AssertionError' at 'MathTest.testMultiply(MathTest.java:27)' with the message '1 \* 1 = 1 expected:<1> but was:<0>'. The failure trace is titled 'Failure Trace' and shows the error details.

Package Explorer JUnit

Finished after 0.023 seconds

Runs: 3/3 Errors: 0 Failures: 1

MathTest [Runner: JUnit 4] (0.000 s)

- testMultiply (0.000 s)
- testAddSimple (0.000 s)
- testAddZero (0.000 s)

Failure Trace

java.lang.AssertionError: 1 \* 1 = 1 expected:<1> but was:<0>  
at MathTest.testMultiply(MathTest.java:27)  
©Manfred Kerber

# JUnit assertions

| Statement                                                | What it does                                                                                       |
|----------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <code>fail()</code>                                      | Lets the test fail. Useful for checking code is not reached under certain conditions.              |
| <code>assertTrue(boolCond)</code>                        | Checks that a condition is true                                                                    |
| <code>assertEquals(expected, actual, [tolerance])</code> | Checks that two values are the same. Not a deep check, i.e., be careful with non-primitive values. |
| <code>assertArrayEquals(expArray, actualArray)</code>    | checks if two arrays are equal.                                                                    |
| <code>assertNull(object)</code>                          | Check that an object is null                                                                       |
| <code>assertNotNull(object)</code>                       | Check that an object is not null                                                                   |
| <code>assertSame(expected, actual)</code>                | Check that both object references are the same.                                                    |
| <code>assertNotSame(expected, actual)</code>             | Check that both object references are not the same.                                                |

- All methods take an optional first argument **message**.
- Last argument tolerance for tests involving floating point numbers.

# JUnit Assertions

```
/* from https://github.com/junit-team/junit/wiki/Assertions */
public class AssertTests {
    @Test
    public void testAssertArrayEquals() {
        byte[] expected = "trial".getBytes();
        byte[] actual = "trial".getBytes();
        assertEquals("failure - arrays not same", expected, actual);
    }

    @Test
    public void testAssertEquals() {
        assertEquals("failure - strings are not equal", "text", "text");
    }

    @Test
    public void testAssertFalse() {
        assertFalse("failure - should be false", false);
    }
}
```

# JUnit Assertions (Cont'd)

```
@Test
public void testAssertNotNull() {
    assertNotNull("should not be null", new Object());
}
```

```
@Test
public void testAssertNotSame() {
    assertNotSame("should not be same Object",
        new Object(), new Object());
}
```

```
@Test
public void testAssertNull() {
    assertNull("should be null", null);
}
```

```
@Test
public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    assertSame("should be same", aNumber, aNumber);
}
```

# JUnit Assertions (Cont'd)

```
// JUnit Matchers assertThat
@Test
public void testAssertThatBothContainsString() {
    assertThat("albumen",
        both(containsString("a")).and(containsString("b")));
}

@Test
public void testAssertThatHasItemsContainsString() {
    assertThat(Arrays.asList("one", "two", "three"),
        hasItems("one", "three"));
}

@Test
public void testAssertThatEveryItemContainsString() {
    assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),
        everyItem(containsString("n")));
}
```



# JUnit Assertions (Cont'd)

@Test

```
public void testAssertThatHamcrestCoreMatchers() {  
    assertThat("good", allOf(equalTo("good"), startsWith("good")));  
    assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));  
    assertThat("good", anyOf(equalTo("bad"), equalTo("good")));  
    assertThat(7, not(CombinableMatcher.<Integer>  
        either(equalTo(3)).or(equalTo(4))));  
    assertThat(new Object(), not(sameInstance(new Object())));  
}
```

@Test

```
public void testAssertTrue() {  
    assertTrue("failure - should be true", true);  
}  
}
```

# JUnit Test for Exception

in Date.java:

```
public Date(int day, String month, int year) {  
    if (admissible(day, month, year)) {  
        this.day = day; ...  
    } else {  
        throw new  
            IllegalArgumentException("Invalid date in Date");  
    }  
}
```

in DateTest.java:

```
@Rule public ExpectedException exception =  
    ExpectedException.none();  
@Test public void dateTest9() {  
    exception.expect(IllegalArgumentException.class);  
    exception.expectMessage("Invalid date in Date");  
    new Date(31, "April", 2016);  
}
```

Test succeeds if correct exception with correct error message is thrown.

# JUnit Annotations

| Statement                             | What it does                                            |
|---------------------------------------|---------------------------------------------------------|
| <code>@Test</code>                    | Marks a test method                                     |
| <code>@Test(timeout=1000)</code>      | Test fails if timeout (in ms) exceeded                  |
| <code>@Test(expException)</code>      | Test fails if defined exception is not thrown           |
| <code>@Before @After</code>           | Code that should be executed before or after every test |
| <code>@BeforeClass @AfterClass</code> | One-off setup/teardown code (e.g. database login)       |
| <code>@Ignore</code>                  | Ignore this test                                        |

Usage:

```
@Test(timeout=1000)
public void underOneSecond() {}
```

# JUnit cannot Test Everything

- Input/output is hard to test (need to maintain separate test files, prepare strings/arrays that simulate file contents if code to be tested supports streams).
- GUIs are even harder to test (separation of GUI and underlying logic helps, e.g. model-view-controller design pattern)

More info online:

- JUnit homepage: <http://junit.org>
- Eclipse JUnit tutorial  
<http://www.vogella.com/articles/JUnit/article.html>

# JUnit cannot Test Everything

- Input/output is hard to test (need to maintain separate test files, prepare strings/arrays that simulate file contents if code to be tested supports streams).
- GUIs are even harder to test (separation of GUI and underlying logic helps, e.g. model-view-controller design pattern)

More info online:

- JUnit homepage: <http://junit.org>
- Eclipse JUnit tutorial  
<http://www.vogella.com/articles/JUnit/article.html>

## Final Point to take home:

- Kiss (Keep it simple, stupid!)
- Design by contract.
- Use design patterns.
- Try to write beautiful code.

# Functions

Unlike methods, functions (also called lambda-expressions) can be called as arguments in methods. Syntax example:

```
import java.util.function.Function;
public class FunMain {
    public static void printN(Function<Integer,Integer> f, int n){
        for (int i = 0; i <= n; i++){
            System.out.print(f.apply(i) + " ");
        }
        System.out.println();
    }
    public static final Function<Integer,Integer> f0 =
        x -> {return x * x + x - 7;};
    public static void main(String[] args) {
        printN(x -> {return x * x;}, 10);
        printN(x -> {return x + 1;}, 10);
        printN(f0, 10);
    }
}
```

Distinguish:

- **Classes**, e.g., BankAccount, Customer
- **Objects**, e.g., bankAccountJohn, customerMary  
created by a **Constructor**, e.g.  
`public BankAccount (Customer customer, String  
password)`
- **Methods**, e.g. `getBalance()`

- Only **one class declaration** in a particular file can be **public**. It corresponds to the file name.
- Classes can be nested. Inner classes are invisible from the outside and corresponding methods cannot be called from the outside.



There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a “**contract**” that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group’s code is written. Generally speaking, interfaces are such contracts.

from `http:`

`//docs.oracle.com/javase/tutorial/java/IandI/createinterface.html`

# Interface (Cont'd)

Allow to apply same method to unrelated objects, e.g. employees and invoices (as expenditure) The example from [Deitel & Deitel, 2010, p. 427 ff.]

```
public interface Payable {
    int getPaymentAmount();
    // no implementation, only head of method
}

public class Invoice implements Payable{
    // implementation of getPaymentAmount()
    public int getPaymentAmount(){
        ...
        return ...;
    }
}

public class Employee implements Payable{
    // implementation of getPaymentAmount()
    public int getPaymentAmount(){
        ...
        return ...;
    }
}
```

