http://www.wakaleo.com
john.smart@wakaleo.com

Testing more efficiently with JUnit 4.4

Course Outline

- Outline
  - Introducing JUnit 4.4
  - Simple JUnit 4.4 tests
  - Fixture methods
  - Handling Exceptions
  - Using Parameterized Tests
  - Using Timeouts
  - Hamcrest asserts
  - JUnit Theories

http://www.wakaleo.com
john.smart@wakaleo.com

Introducing JUnit 4.4

Introducing JUnit 4.4

- From JUnit 3.x to JUnit 4.4

JUnit 3.x	JUnit 4.x
All classes derive from **TestCase**	Any class can contain tests
setUp() and **tearDown()**	**@before**, **@beforeClass**, **@after**, **@afterClass**
Tests must be called **testXXX()**	Tests use the **@Test** annotation
	Timeouts
	Testing Exceptions
	Parameterized Tests
	Theories

Introducing JUnit 4.4

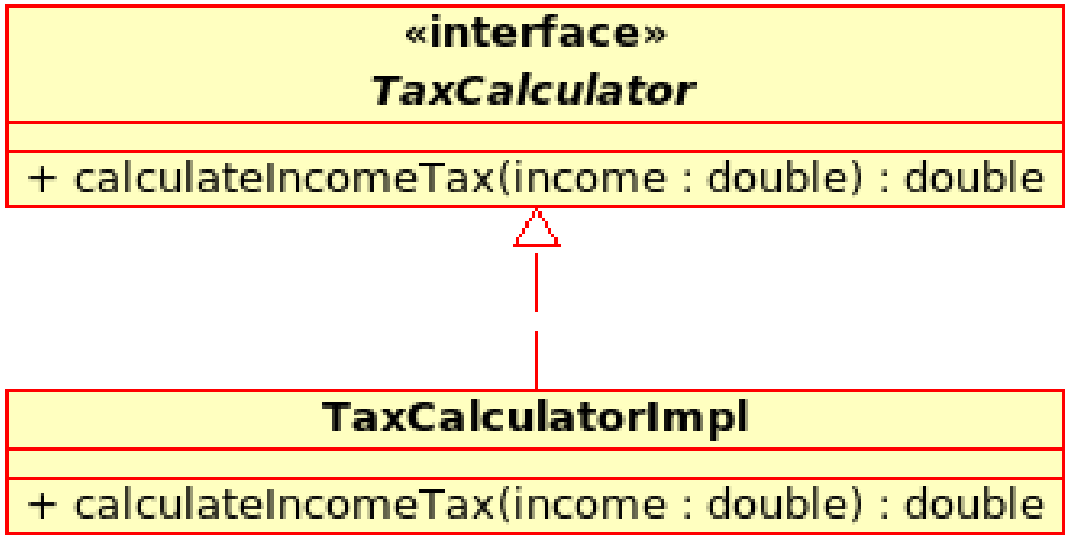
- Testing with JUnit 4.4
  - Case study: **A Tax Calculator**

Business Rule #1:

Income up to $38,000 is taxed at 19.5%

Introducing JUnit 4.4

- Testing with JUnit 4.4
  - The classes being tested:



Introducing JUnit 4.4

- Testing with JUnit 4.4
  - The class being tested:

```
public interface TaxCalculator {
    double calculateIncomeTax(double income);
}
```

```
public class TaxCalculatorImpl implements TaxCalculator {

    @Override
    public double calculateIncomeTax(double income) {
        ...
    }

}
```

Introducing JUnit 4.4

- Testing with JUnit 4.4
  - The first business rule:
  - Our first u.

Income up to $38,000 is taxed at 19.5%

Does not derive from TestClass

@Test indicates a unit test

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

public class TaxCalculatorImplTest {

    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        TaxCalculatorImpl taxCalculator = new TaxCalculatorImpl();

        double income = 30000;
        double expectedTax = income * 0.195;
        double calculatedTax = taxCalculator.calculateIncomeTax(30000);
        assertEquals("Tax below 38000 should be taxed at 19.5%",
                     expectedTax, calculatedTax, 0);
    }
}
```

Unit test method name doesn't have to start with “test”

http://www.wakaleo.com
john.smart@wakaleo.com

Fixture Methods

Introducing JUnit 4.4
Fixture methods
Handling Exceptions
Using Parameterized Tests
Using Timeouts
Hamcrest asserts
JUnit Theories

Fixture methods

- Setting up your tests, and tidying up afterwards
  - In JUnit 3.x, you had **setUp()** and **tearDown()**
  - In JUnit 4.x, you have:
    - **@BeforeClass -** run before any test has been executed
    - **@Before** – run before each test.
    - **@After** – run after each test
    - **@AfterClass** – run after all the tests have been executed

Fixture methods

- Setting up your tests

Business Rule #2:

Losses should not be taxed.

Fixture methods

- Setting up your tests
  - Using the **@Before** annotation

Losses should not be taxed.

```
public class TaxCalculatorImplTest {

    TaxCalculatorImpl taxCalculator = null;

    @Before
    public void prepareTaxCalculator() {
        taxCalculator = new TaxCalculatorImpl();
    }

    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        double income = 30000;
        double expectedTax = income * 0.195;
        double calculatedTax = taxCalculator.calculateIncomeTax(30000);
        assertEquals("Tax below 38000 should be taxed at 19.5%", expectedTax, calculatedTax, 0);
    }

    @Test
    public void lossesShouldNotBeTaxed() {
        double calculatedTax = taxCalculator.calculateIncomeTax(-10000);
        assertEquals("Losses should not be taxed", 0, calculatedTax, 0);
    }
}
```

Executed *before* each test

Fixture methods

- Tidying up afterwards
  - Using the **@After** annotation

```
public class TaxCalculatorImplTest {

    TaxCalculatorImpl taxCalculator = null;

    @Before
    public void prepareTaxCalculator() {
        taxCalculator = new TaxCalculatorImpl();
    }

    @After
    public static void tidyUp() {
        taxCalculator = null;
     }

    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        double income = 30000;
        double expectedTax = income * 0.195;
        double calculatedTax = taxCalculator.calculateIncomeTax(30000);
        assertEquals("Tax below 38000 should be taxed at 19.5%", expectedTax, calculatedTax, 0);
    }
    ...
}
```

Executed *after* each test

Fixture methods

- Setting up your test suite
  - Using the **@BeforeClass** annotation

Executed once before any test is executed.
The method must be static

```
public class TaxCalculatorImplTest {

    static TaxCalculatorImpl taxCalculator = null;

    @BeforeClass
    public static void prepareTaxCalculator() {
        taxCalculator = new TaxCalculatorImpl();
    }

    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        double income = 30000;
        double expectedTax = income * 0.195;
        double calculatedTax = taxCalculator.calculateIncomeTax(30000);
        assertEquals("Tax below 38000 should be taxed at 19.5%", expectedTax, calculatedTax, 0);
    }
    ...
}
```

Fixture methods

- Tidying up after your test suite
  - Using the **@AfterClass** annotation

Executed once before *any test* is executed. The method must be static

```
public class TaxCalculatorImplTest {

    static TaxCalculatorImpl taxCalculator = null;

    @BeforeClass
    public static void prepareTaxCalculator() {
        taxCalculator = new TaxCalculatorImpl();
    }

    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        double income = 30000;
        double expectedTax = income * 0.195;
        double calculatedTax = taxCalculator.calculateIncomeTax(30000);
        assertEquals("Tax below 38000 should be taxed at 19.5%", expectedTax, calculatedTax, 0);
    }

    @Test
    public void lossesShouldNotBeTaxed() {
        double calculatedTax = taxCalculator.calculateIncomeTax(-10000);
        assertEquals("Losses should not be taxed", 0, calculatedTax, 0);
    }
}
```

Fixture methods

- Tidying up afterwards
  - Using the **@After** annotation

```
public class TaxCalculatorImplTest {

    static TaxCalculatorImpl taxCalculator = null;

    @BeforeClass
    public static void prepareTaxCalculator() {
        taxCalculator = new TaxCalculator
    }

    @AfterClass
    public static void tidyUp() {
        taxCalculator= null;
    }

    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        double income = 30000;
        double expectedTax = income * 0.195;
        double calculatedTax = taxCalculator.calculateIncomeTax(30000);
        assertEquals("Tax below 38000 should be taxed at 19.5%", expectedTax, calculatedTax, 0);
    }...
}
```

Executed once after *every test* has been executed.
The method must be static

http://www.wakaleo.com
john.smart@wakaleo.com

Handling Exceptions

Introducing JUnit 4.4
Fixture methods
Handling Exceptions
Using Parameterized Tests
Using Timeouts
Hamcrest asserts
JUnit Theories

Handling Exceptions

- Testing for expected Exceptions
  - Use the *expected* parameter of the **@Test** annotation

Handling Exceptions

- Testing for excepted Exceptions
  - A practical example: income tax rates can change each year. So we need to specify the year in our TaxCalculator.
  - If an invalid year is provided, the class throws an InvalidYearException.

Business Rule #3:

The tax year cannot be in the future.

Handling Exceptions

- Testing for excepted Exceptions
  - The TaxCalculator interface now looks like this:

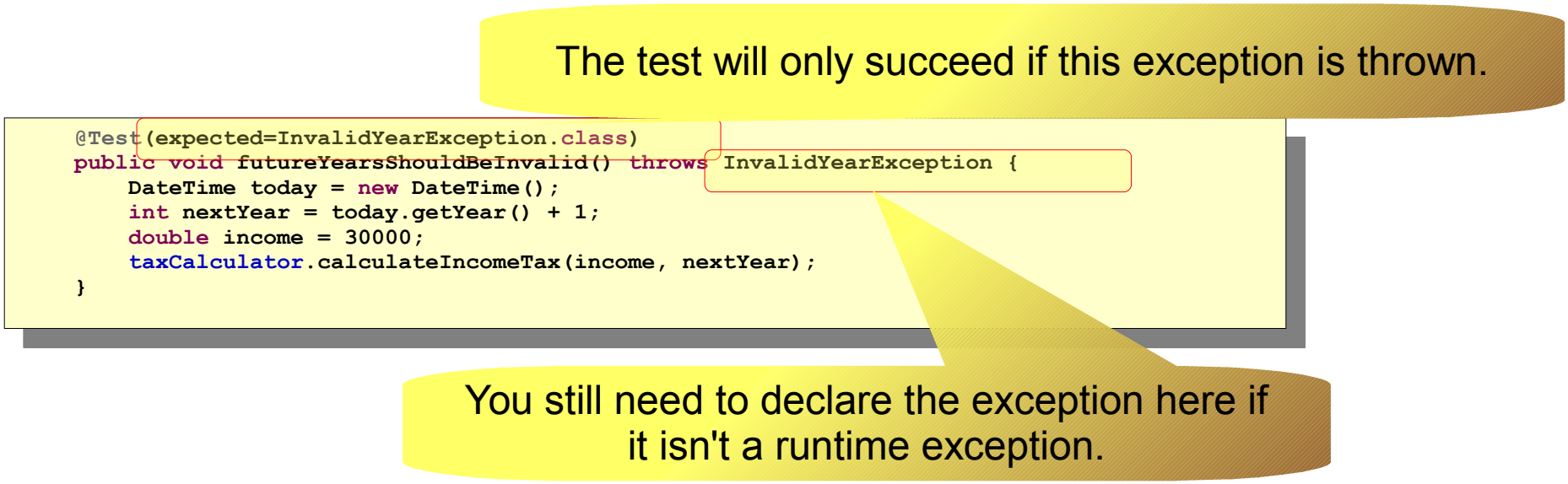
```
public interface TaxCalculator {
    double calculateIncomeTax(double income, int year) throws InvalidYearException;
}
```

Now we also provide the year

If the year is invalid, throw an InvalidYearException

Handling Exceptions

- Testing for excepted Exceptions
  - Using the **expected** parameter
    - A simple way to test that an Exception is thrown



Handling Exceptions

- Limitations of this approach
  - The traditional approach is better for:
    - Running assertions against the exception

      e.g. Checking the Exception message

```
@Test
public void exceptionShouldIncludeAClearMessage() throws InvalidYear...ception {
    try {
        taxCalculator.calculateIncomeTax(50000, 2100);
        fail("calculateIncomeTax() should have thrown an exception.");
    } catch (InvalidYearException expected) {
        assertEquals(expected.getMessage(),
                     "No tax calculations available yet for the year 2100");
    }
}
```

Make sure the exception was thrown

Check the message in the Exception

Handling Exceptions

- Limitations of this approach
  - The traditional approach is better for:
    - Checking application state *after* the exception

      e.g. Withdrawing money from a bank account

```
@Test
public void failedWithdrawlShouldNotDebitAccount(){
    Account account = new Account();
    account.setBalance(100);
    try {
        account.withdraw(200);
        fail("withdraw() should have thrown an InsufficantFundsException.");
    } catch (InsufficantFundsException e) {
        assertEquals("Account should not have been debited",
                     100.0,account.getBalance(),0.0);
    }
}
```

Make sure the exception was thrown

Verify the state of the account afterwards

http://www.wakaleo.com
john.smart@wakaleo.com

Using Parameterized Tests

Introducing JUnit 4.4
Fixture methods
Handling Exceptions
Using Parameterized Tests
Using Timeouts
Hamcrest asserts
JUnit Theories

Using Parameterized Tests

- Parametrized tests:
  - Run several sets of test data against the same test case
  - Help reduce the number of unit tests to write
  - Encourage developers to test more thoroughly

Using Parameterized Tests

- Parametrized tests:
  - Example: Calculating income tax

Taxable Income	Tax rate
up to $38,000	19.5 cents
$38,001 to $60,000 inclusive	33 cents
$60,001 and over	39 cents

Using Parameterized Tests

- Parametrized tests
  - What you need:

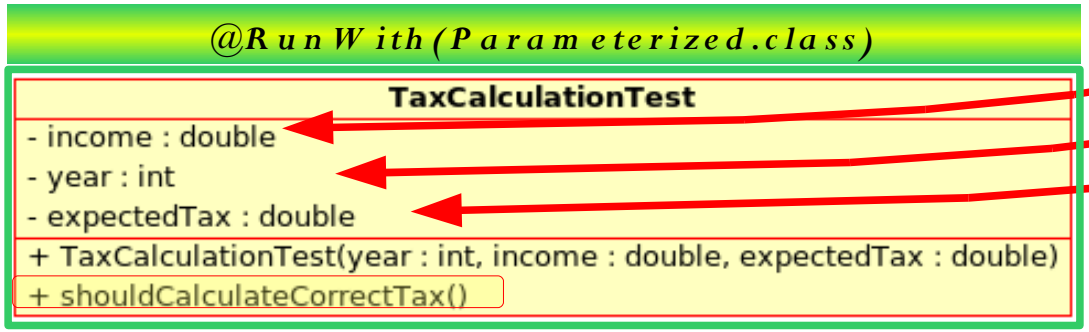
Some test data

A test class with matching fields...

and some tests

and an annotation

Income	Year	Expected Tax
$0.00	2007	$0.00
$10,000.00	2007	$1,950.00
$20,000.00	2007	$3,900.00
$38,000.00	2007	$7,410.00
$38,001.00	2007	$7,410.33
$40,000.00	2007	$8,070.00
$60,000.00	2007	$14,670.00
$100,000.00	2007	$30,270.00



Using Parameterized Tests

- Parameterized tests
  - How does it work?

This is a parameterized test

The @Parameters annotation indicates the test data.

```
@RunWith(Parameterized.class)
public class TaxCalculationTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
        /* Income Year Tax */
        { 0.00, 2006, 0.00 }, { 10000.00, 2006, 1950.00 },
        { 20000.00, 2006, 3900.00 }, { 38000.00, 2006, 7410.00 },
        { 38001.00, 2006, 7410.33 }, { 40000.00, 2006, 8070.00 },
        { 60000.00, 2006, 14670.00 }, { 100000.00, 2006, 30270.00 }, });
    }

    private double income;
    private int year;
    private double expectedTax;

    public TaxCalculationTest(double income, int year, double expectedTax) {
        this.income = income;
        this.year = year;
        this.expectedTax = expectedTax;
    }

    @Test
    public void shouldCalculateCorrectTax() throws InvalidYearException {
        TaxCalculator calculator = new TaxCalculatorImpl();
        double calculatedTax = calculator.calculateIncomeTax(income, year);
        assertEquals(expectedTax, calculatedTax, 0.0);
    }
}
```

Income	Year	Expected Tax
$0.00	2007	$0.00
$10,000.00	2007	$1,950.00
$20,000.00	2007	$3,900.00
$38,000.00	2007	$7,410.00
$38,001.00	2007	$7,410.33
$40,000.00	2007	$8,070.00
$60,000.00	2007	$14,670.00
$100,000.00	2007	$30,270.00

The constructor takes the fields from the test data

The unit tests use data from these fields.

Using Parameterized Tests

- Parametrized tests
  - The **@RunWith** annotation and the Parameterized runner

```
@RunWith(Parameterized.class)
public class TaxCalculationTest {
...
```

Tells Junit to run this class as a parameterized test case

Using Parameterized Tests

- Parametrized tests
  - The **@Parameters** annotation and the test data

The test data is a 2-dimentional array

```
@RunWith(Parameterized.class)
public class TaxCalculationTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
        /* Income Year Tax */
        { 0.00, 2006, 0.00 }, { 10000.00, 2006, 1950.00 },
        { 20000.00, 2006, 3900.00 }, { 38000.00, 2006, 7410.00 },
        { 38001.00, 2006, 7410.33 }, { 40000.00, 2006, 8070.00 },
        { 60000.00, 2006, 14670.00 }, { 100000.00, 2006, 30270.00 }, });
    }
...
```

Using Parameterized Tests

- Parametrized tests
  - The member variables and the constructor

A member variable for each element of test data

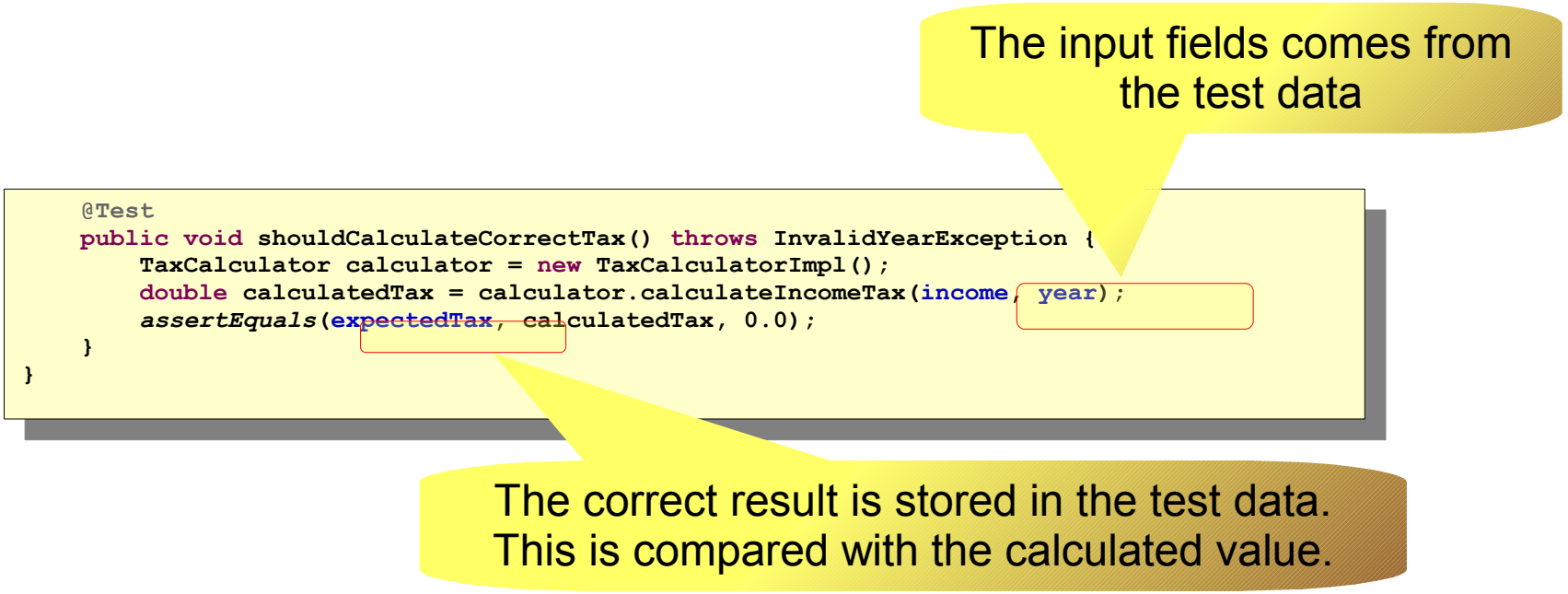
JUnit initialises instances of this class by passing rows of test data to this constructer

```
...
private double income;
private int year;
private double expectedTax;

public TaxCalculationTest(double income, int year, double expectedTax) {
    this.income = income;
    this.year = year;
    this.expectedTax = expectedTax;
}
...
```

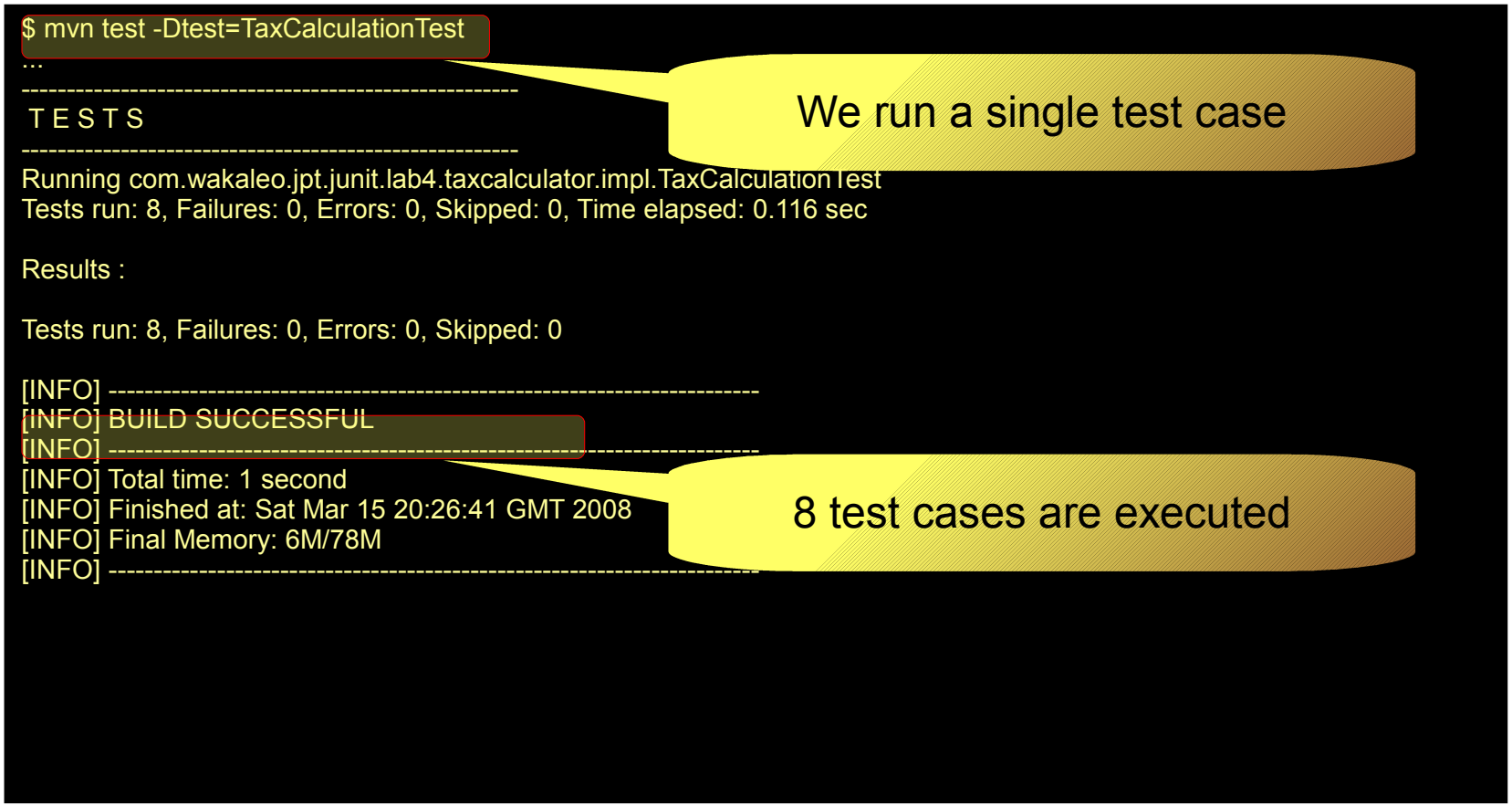
Using Parameterized Tests

- The unit tests
  - The unit tests use the member variables to check the tested class.



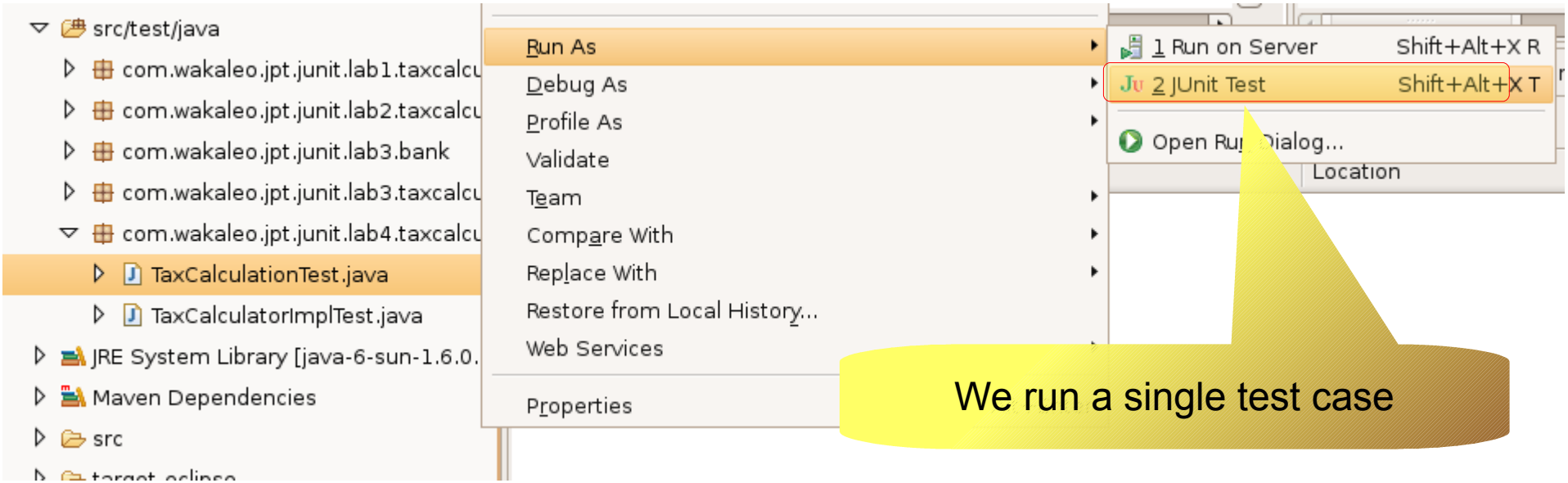
Using Parameterized Tests

- Running parameterized tests in Maven



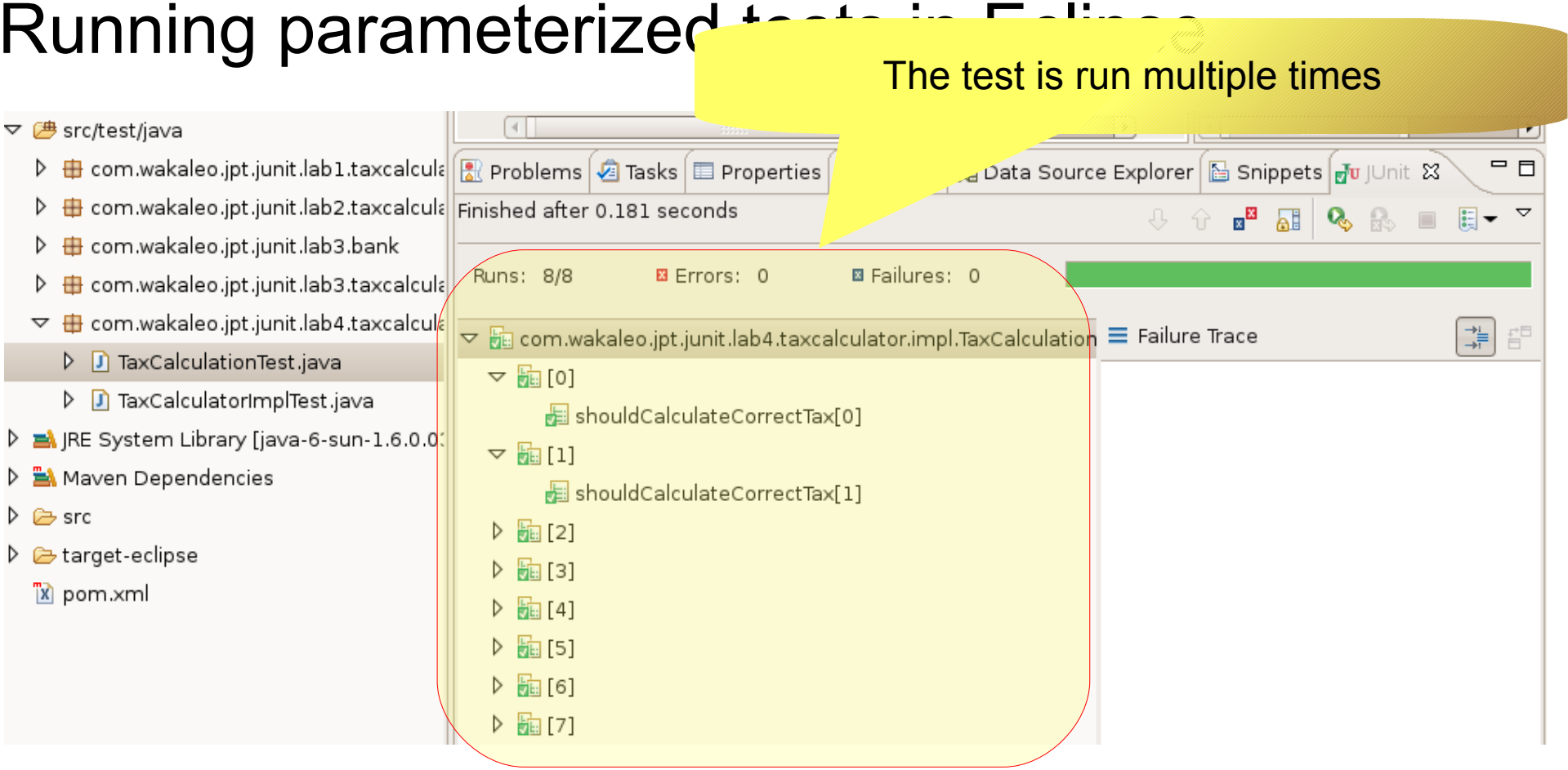
Using Parameterized Tests

- Running parameterized tests in Eclipse



Using Parameterized Tests

- Running parameterized tests in Eclipse



http://www.wakaleo.com
john.smart@wakaleo.com

Using Timeouts

Introducing JUnit 4.4
Fixture methods
Handling Exceptions
Using Parameterized Tests
Using Timeouts
Hamcrest asserts
JUnit Theories

Using Timeouts

- Simple performance tests
  - Use the *timeout* parameter of the **@Test** annotation

```
@Test(timeout=100)
public void shouldCalculateCorrectTax() throws InvalidYearException {
    TaxCalculator calculator = new TaxCalculatorImpl();
    double calculatedTax = calculator.calculateIncomeTax(income, year);
    assertEquals(expectedTax, calculatedTax, 0.0);
}
```

Test will fail after 100 ms

Using Timeouts

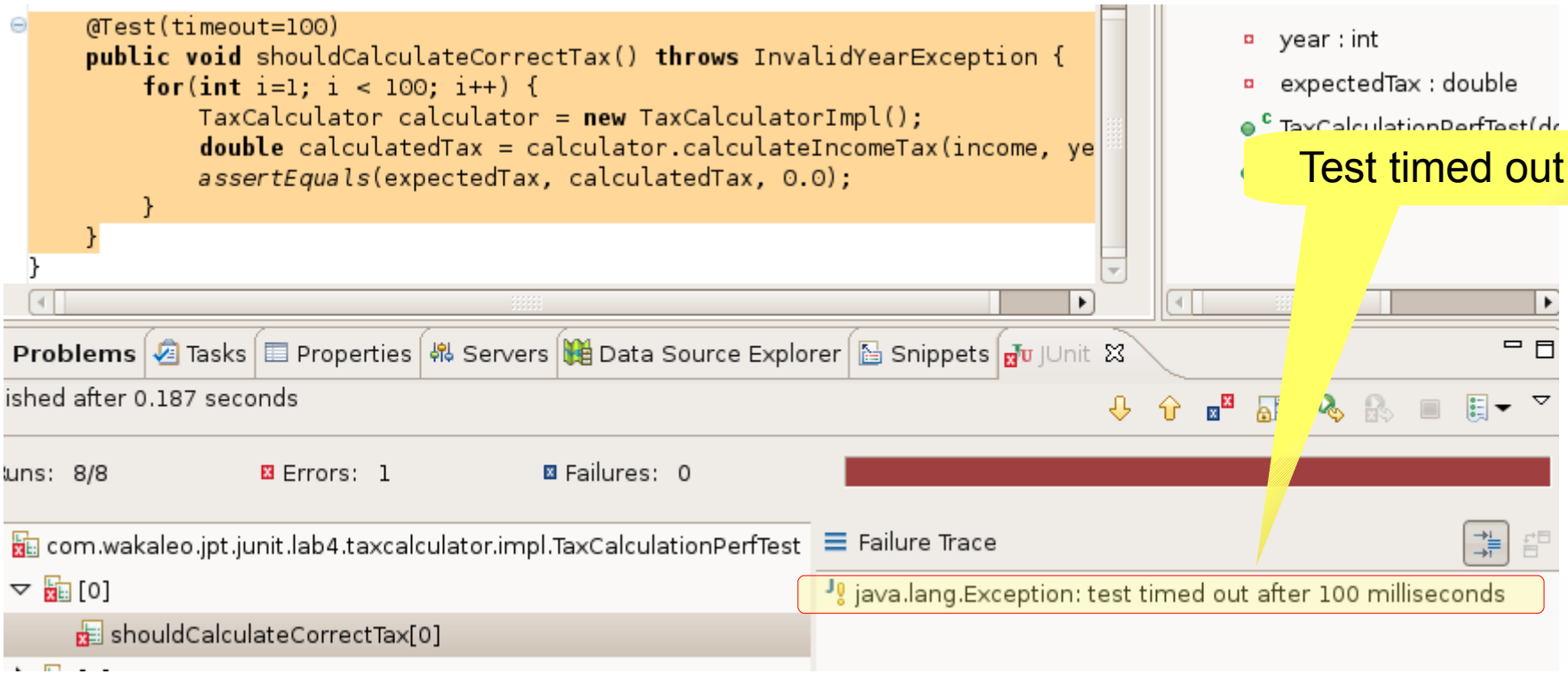
- Simple performance tests
  - Sometimes, you need to repeat operations for good results...

```
@Test(timeout=1000)
public void shouldCalculateCorrectTax() throws InvalidYearException {
    for(int i=1; i < 50; i++) {
        TaxCalculator calculator = new TaxCalculatorImpl();
        double calculatedTax = calculator.calculateIncomeTax(income, year);
        assertEquals(expectedTax, calculatedTax, 0.0);
    }
}
```

Time 50 calculations for more realistic results

Using Timeouts

- Simple performance tests
  - The test will fail if it takes more than the specified time



Using Timeouts

- Integration tests, not Unit tests
  - Use with care:
    - Run them with your integration tests, not with your unit tests
    - If your criteria are too demanding, they may fail unexpectedly:
      - Machine load
      - Processor speed
      - ...

http://www.wakaleo.com
john.smart@wakaleo.com

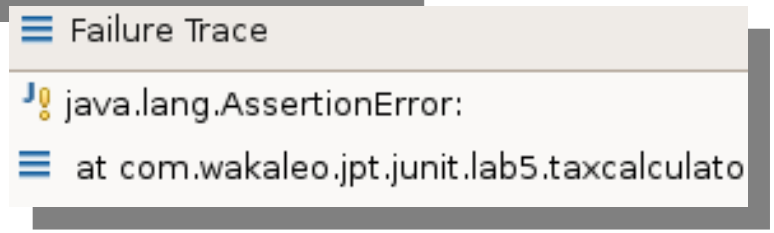
Hamcrest asserts

Introducing JUnit 4.4
Fixture methods
Handling Exceptions
Using Parameterized Tests
Using Timeouts
Hamcrest asserts
JUnit Theories

Hamcrest asserts

- Traditional JUnit 3.x asserts are hard to read:
  - Parameter order is counter-intuitive for English-speakers
    - x=10 is written `assertEquals(10, x);`
  - The statements don't read well for English-speakers
    - “Assert that are equal 10 and x”
  - Default error messages are sometimes limited:

```
String color = "yellow";
assertTrue(color.equals("red") || color.equals("blue") );
```



Hamcrest asserts

- JUnit 4.4 introduces the **assertThat** statement
  - Rather than writing:

```
import static org.junit.Assert.*;
...
assertEquals(expectedTax, calculatedTax, 0);
```

  - You can write

```
import static org.hamcrest.Matchers.*;
...
assertThat(calculatedTax, is(expectedTax));
```

Hamcrest asserts

- Advantages of the **assertThat** statement:
  - More readable and natural assertions

```
assertThat(calculatedTax, is(expectedTax));
```

  - Readable failu

"Assert that calculated tax is [the same as] expected tax"

  -

```
String color = "red";
assertThat(color, is("blue"));
```

Failure Trace

java.lang.AssertionError:

Expected: is "blue"

got: "red"

```
String[] colors = new String[] {"red","green","blue"};
String color = "yellow";
assertThat(color, not(isIn(colors)));
```

Hamcrest asserts

- Simple equality tests

Assert that...is

```
String color = "red";
assertThat(color, is("red"));
```

Assert that...equalTo

```
String color = "red";
assertThat(color, equalTo("red"));
```

Assert that...not

```
String color = "red";
assertThat(color, not("blue"));
```

Hamcrest asserts

- More sophisticated equality tests

Assert that...is one of

```
String color = "red";
assertThat(color, isOneOf("red","blue","green"));
```

Assert that...is a class

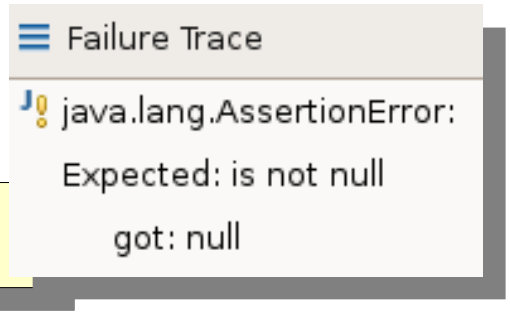
```
List myList = new ArrayList();
assertThat(myList,is(Collection.class));
```

Hamcrest asserts

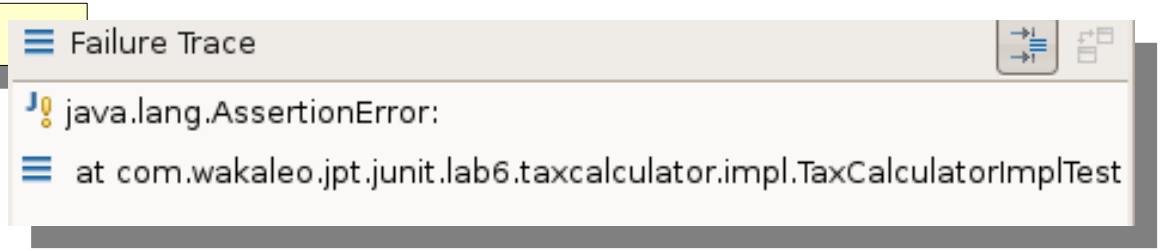
- Testing for null values

notNullValue()

```
String color = "red";
assertThat(color, is(notNullValue()));
```




```
assertNotNull(color);
```



nullValue()

```
String color = null;
assertThat(color, is(nullValue()));
```

```
assertNull(color);
```

Hamcrest asserts

- Testing with collections

hasItem()

```
List<String> colors = new ArrayList<String>();
colors.add("red");
colors.add("green");
colors.add("blue");
assertThat(colors, hasItem("blue"));
```

hasItems()

```
assertThat(colors, hasItems("red","green"));
```

hasItemsInArray()

```
String[] colors = new String[] {"red","green","blue"};
assertThat(colors, hasItemInArray("blue"));
```

Hamcrest asserts

- Testing with collections

hasValue()

```
Map map = new HashMap();
map.put("color", "red");
assertThat(map, hasValue("red"));
```

Combined matchers

```
List<Integer> ages = new ArrayList<Integer>();
ages.add(20);
ages.add(30);
ages.add(40);
assertThat(ages, not(hasItem(lessThan(18))));
```

"This list does not have an item that is less than 18"

Hamcrest asserts

- But don't go overboard...
  - Which is better? This?

```
int value = 15;
assertThat(value, allOf(greaterThanOrEqualTo(10),
                        lessThanOrEqualTo(20)));
```

or this?

```
int value = 15;
assertTrue("Value should be between 10 and 20",
           value >= 10 && value <= 20);
```

http://www.wakaleo.com
john.smart@wakaleo.com

JUnit Theories

Introducing JUnit 4.4
Fixture methods
Handling Exceptions
Using Parameterized Tests
Using Timeouts
Hamcrest asserts
JUnit Theories

JUnit 4 Theories

- JUnit 4 theories – testing multiple data sets
  - With JUnit 4 theories, you can
    - Test large combinations of data values in a single method
    - Document what you think your code should do a little better
  - A theory is a statement that is true for many data sets.
    - Instead of “shouldDoThis” or “shouldDoThat”, we say “isThis” or “isThat”.

JUnit 4 Theories

- A simple example – testing tax rates
  - 2007 and 2008 Tax Rates

Taxable Income	Tax rate
up to $38,000	19.5 cents
$38,001 to $60,000 inclusive	33 cents
$60,001 and over	39 cents

In 2007 and 2008, income up to $38 000 is taxed at 19.5%

JUnit 4 Theories

- A simple example – the business rule

In 2007 and 2008, income up to $38 000 is taxed at 19.5%

  - This becomes our theory. We express it as ...

Test data will be injected here

We use the **@Theory** annotation

```
@Theory
public void incomeUpTo38000IsTaxedAtLowestRate(double income,
                                               int year) {
    assumeThat(year,anyOf(is(2007),is(2008)));
    assumeThat(income,lessThanOrEqualTo(38000.00));
    TaxCalculator calculator = new TaxCalculatorImpl();
    double calculatedTax
              = calculator.calculateIncomeTax(income, year);
    double expectedTax = income * 0.195;
    assertThat(expectedTax,is(calculatedTax));
}
```

Only applies for 2007 and 2008 and for incomes that are under 38000

What do we expect?

Does it match?

JUnit 4 Theories

- JUnit 4 theories – some explanations
  - **@Theory** – declares a method that tests a theory.
  - A **@Theory** method has parameters, which are used to inject test data.
  - **assumeThat()** - Hamcrest-style expression used to filter out the test data values that you are interested in for that theory.

JUnit 4 Theories

- A simple example – and now for some test data
  - Test data is indicated by the **@DataPoint** annotation:

```
@DataPoint public static int YEAR_2008 = 2008;
```

  - Datapoints are public static variables of different types:

```
@DataPoint public static int YEAR_2007 = 2007;
@DataPoint public static int YEAR_2008 = 2008;
@DataPoint public static double INCOME_1 = 0.0;
@DataPoint public static double INCOME_2 = 1000.0;
@DataPoint public static double INCOME_3 = 5000.0;
...
```

JUnit 4 Theories

- A simple example – and now for some test data
  - Datapoint values are injected into the theory methods according to their type:

```
@DataPoint public static int YEAR_2007 = 2007;
@DataPoint public static int YEAR_2008 = 2008;
@DataPoint public static double INCOME_1 = 0.0;
@DataPoint public static double INCOME_2 = 1000.0;
@DataPoint public static double INCOME_3 = 5000.0;
...
```

```
@Theory
public void incomeUpTo38000IsTaxedAtLowestRate(double income, int year) {
    ...
}
```

JUnit 4 Theories

- Datapoints – some explanations
  - **@Datapoint** values are injected into the **@Theory** methods according to their type.
    - E.g. If you have 20 integer data points, they will all be sent to every integer @Theory method parameter.
  - You use **assumeThat()** expressions to filter out the values that you aren't interested in.
  - You should apply **assumeThat()** expressions to every parameter.

JUnit 4 Theories

- A simple example – the full test case

```
@RunWith(Theories.class)
public class TaxCalculationTheoryTest {

    @DataPoint public static int YEAR_2006 = 2006;
    @DataPoint public static int YEAR_2007 = 2007;
    @DataPoint public static int YEAR_2008 = 2008;
    @DataPoint public static double INCOME_1 = 0;
    @DataPoint public static double INCOME_2 = 1000;
    ...
    @DataPoint public static double INCOME_10= 38000;
    @DataPoint public static double INCOME_14= 50000;
    @DataPoint public static double INCOME_15= 60000;

    @Theory
    public void incomeUpTo38000IsTaxedAtLowestRate(double income,int year) {
        assumeThat(year,anyOf(is(2007),is(2008)));
        assumeThat(income,lessThanOrEqualTo(38000.00));
        TaxCalculator calculator = new TaxCalculatorImpl();
        double calculatedTax = calculator.calculateIncomeTax(income, year);
        double expectedTax = income * 0.195;
        System.out.println("year = " + year
                           + ", income=" + income
                           + ", calculated tax=" + calculatedTax);
        assertThat(expectedTax,is(calculatedTax));
    }
}
```

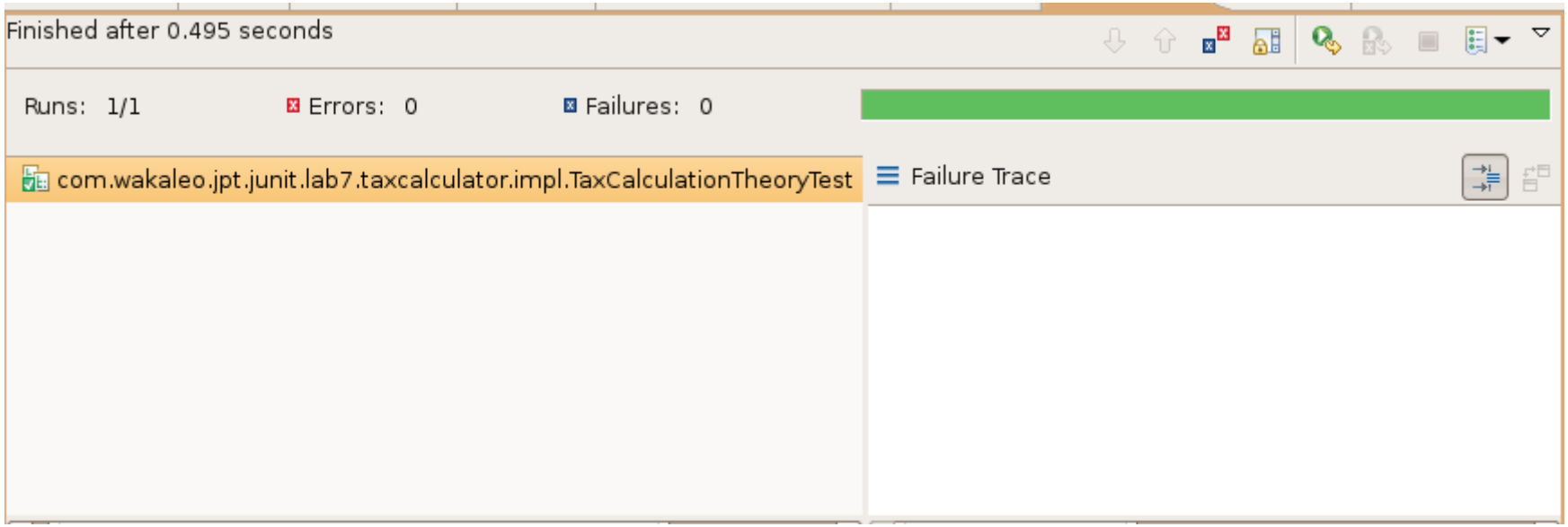
Test data (as much as you like)

Theory relating to this test data

Log message for convenience

JUnit 4 Theories

- A simple example – running the tests
  - Eclipse will run a single test for each @Theory method



JUnit 4 Theories

- A simple example – running the tests
  - However the @Theory will actually be executed once for each combination of corresponding datapoint values, e.g.

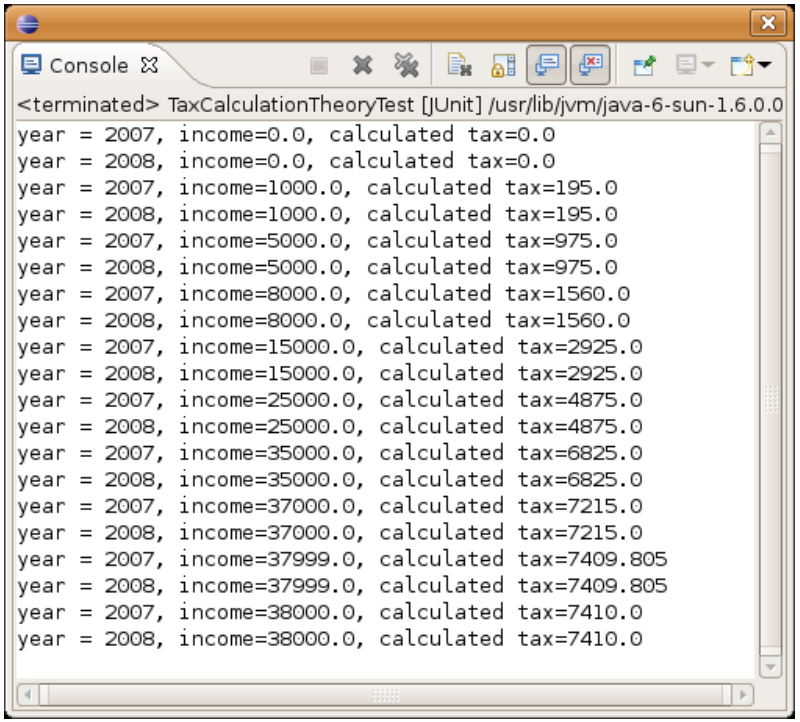
```
@DataPoint public static double INCOME_1 = 0;
@DataPoint public static double INCOME_2 = 1000;
@DataPoint public static double INCOME_3 = 5000;
@DataPoint public static double INCOME_4 = 8000;
@DataPoint public static double INCOME_5 = 15000;
@DataPoint public static double INCOME_6 = 25000;
@DataPoint public static double INCOME_7 = 35000;
@DataPoint public static double INCOME_8 = 37000;
@DataPoint public static double INCOME_9 = 37999;
@DataPoint public static double INCOME_10= 38000;
@DataPoint public static double INCOME_12= 38001;
@DataPoint public static double INCOME_13= 40000;
@DataPoint public static double INCOME_14= 50000;
@DataPoint public static double INCOME_15= 60000;
```

X

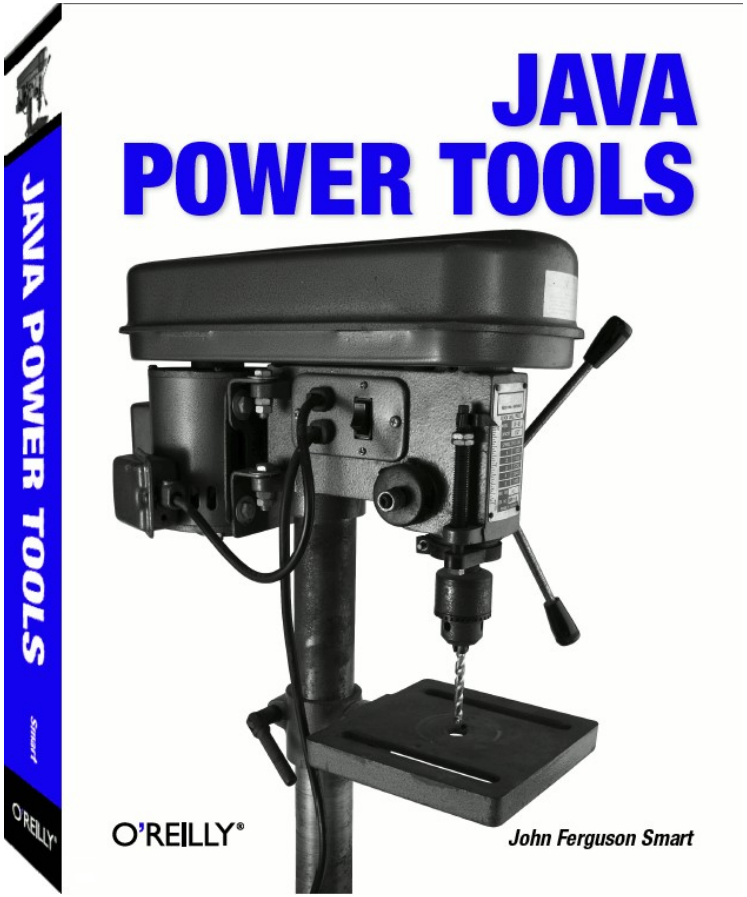
```
@DataPoint public static int YEAR_2006 = 2006;
@DataPoint public static int YEAR_2007 = 2007;
@DataPoint public static int YEAR_2008 = 2008;
```

JUnit 4 Theories

- A simple example – running the tests
  - However the @Theory will actually be executed once for each combination of corresponding datapoint values, e.g.

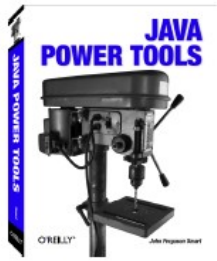


To learn more...



Wakaleo Consulting
Optimizing your software development process
http://www.wakaleo.com

The Java Power Tools Bootcamp



Code better - Code faster - Code smarter

The Java Power Tools Bootcamp is a comprehensive, innovative and hands-on workshop covering best-of-breed open source tools and techniques for Agile Development in Java. Learn how to optimize your development process, hone your programming skills and know-how, and ultimately produce better software. And have fun while you're doing it!

Course Objectives

Students will come away from this workshop with a solid understanding of how they can improve their development practices back in the real world, as well as an abundance of pratical tips and tricks that they can use in their day-to-day work. Notably, after the course, students will:

- Have a practical understanding and experience of Maven 2, and be able to determine for themselves if it is suitable for their project or organisation.
- Understand the issues around dependency management in Java development, and be able to implement declarative dependency management in a corporate environment using both Maven and Ant.
- Know how to write effective unit tests, and understand how to use unit testing practices to write more reliable code faster.
- Be able to write automated database and web interface tests.
- Understand how to use code quality and test coverage metrics to improve your code, and understand what the various metrics can tell you, and also what they can't.
- Have a solid working knowledge of Subversion in the real world.
- Know how to set up a working Continuous Integration server, complete with automated builds, tests, code quality audits and reports, and automatic deployment to an integration server

Thank you

- Questions?