Agenda

- What will we cover today
  - Naming your tests
  - Hamcrest Assertions and Matchers
  - Parameterized tests and JUnit Theories
  - JUnit Rules
  - Parallel Testing
  - JUnit Categories
  - Continuous Testing
  - Mockito

Anatomy of a JUnit 4 test

What's the big deal with JUnit 4?

```
import static com.wakaleo.gameoflife.domain.Cell.DEAD_CELL;
import static com.wakaleo.gameoflife.domain.Cell.LIVE_CELL;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import org.junit.Test;

public class WhenYouCreateACell {

    @Test
    public void aLiveCellShouldBeRepresentedByAnAsterisk() {
        Cell cell = Cell.fromSymbol("*");
        assertThat(cell, is(LIVE_CELL));
    }

    @Ignore("Pending Implementation")
    @Test
    public void aDeadCellShouldBeRepresentedByADot() {
        Cell cell = Cell.fromSymbol(".");
        assertThat(cell, is(DEAD_CELL));
    }
    ...
}
```

No need to extend TestCase

Annotation-based

Call the tests anything you want

Annotations for test metadata

What’s in a name

- Name your tests well

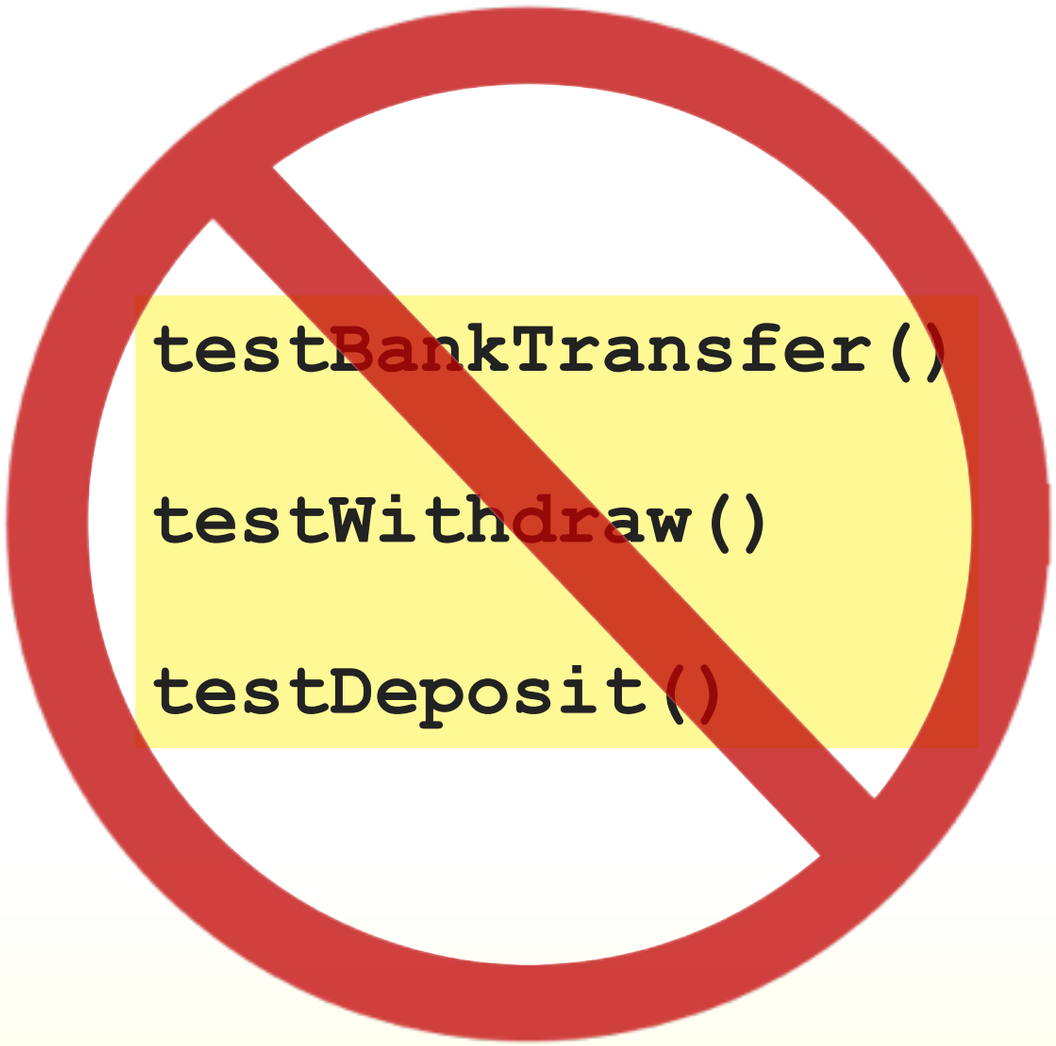
"*What's in a name? That which we call a rose*
By any other name would smell as sweet."
Romeo and Juliet (II, ii, 1-2)

What's in a name

- The ~~10~~ 5 Commandments of Test Writing
  I. Don't say "test, say "should" instead
  II. Don't test your classes, test their behaviour
  III. Test class names are important too
  IV. Structure your tests well
  V. Tests are deliverables too

What’s in a name

- Don’t use the word ‘test’ in your test names



What's in a name

- *Do* use the word 'should' in your test names

```
testBankTransfer()

testWithdraw()
```

```
tranferShouldDeductSumFromSourceAccountBalance()

transferShouldAddSumLessFeesToDestinationAccountBalance()

depositShouldAddAmountToAccountBalance()
```

What's in a name

- Your test class names should represent context

When is this behaviour applicable?

- WhenYouCreateACell
  - aDeadCellShouldBePrintedAsADot() : void
  - aDeadCellShouldBeRepresentedByADot() : void
  - aDeadCellSymbolShouldBeADot() : void
  - aLiveCellShouldBePrintedAsAnAsterisk() : void
  - aLiveCellShouldBeRepresentedByAnAsterisk() : void
  - aLiveCellSymbolShouldBeAnAsterisk() : void

What behaviour are we testing?

What's in a name

- Write your tests consistently
  - 'Given-When-Then' or 'Arrange-Act-Assert' (AAA)

```
@Test
public void aDeadCellWithOneLiveNeighbourShouldRemainDeadInTheNextGeneration() {
    String initialGrid = "...\n" +
                         ".*.\n" +
                         "...";

    String expectedNextGrid = "...\n" +
                              "...\n" +
                              "...\n";

    Universe theUniverse = new Universe(seededWith(initialGrid));

    theUniverse.createNextGeneration();
    String nextGrid = theUniverse.getGrid();

    assertThat(nextGrid, is(expectedNextGrid));
}
```

Prepare the test data ("arrange")

Do what you are testing ("act")

Check the results ("assert")

What's in a name

- Tests are deliverables too - respect them as such
  - Refactor, refactor, refactor!
  - Clean and readable

Express Yourself with Hamcrest

- Why write this...

```
import static org.junit.Assert.*;
...
assertEquals(10000, calculatedTax, 0);
```

- when you can write this...

```
import static org.hamcrest.Matchers.*;
...
assertThat(calculatedTax, is(10000));
```

"Assert that are equal 10000 and calculated tax (more or less)" ?!

Don't I just mean "assert that calculated tax is 10000"?

Express Yourself with Hamcrest

- With Hamcrest, you can have your cake and eat it!

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

Readable asserts

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

Failure Trace

java.lang.AssertionError:

Expected: is "blue"

got: "red"

Informative errors

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

Flexible notation

Express Yourself with Hamcrest

More Hamcrest expressiveness

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

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

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

```
assertThat(colors, hasItem(anyOf(is("red"), is("green"), is("blue"))));
```

Home-made Hamcrest Matchers

- Customizing and extending Hamcrest
  - Combine existing matchers
  - Or make your own!

Home-made Hamcrest Matchers

- Customizing Hamcrest matchers
  - You can build your own by combining existing Matchers...

Create a dedicated Matcher for the Stakeholder class

```
List stakeholders  = stakeholderManager.findByName("Health");
Matcher<Stakeholder> calledHealthCorp = hasProperty("name", is("Health Corp"));
assertThat(stakeholders, hasItem(calledHealthCorp));
```

Use matcher directly with hasItem()

“The stakeholders list has (at least) one item with the name property set to “Health Corp””

Home-made Hamcrest Matchers

- Writing your own matchers in three easy steps!

```
public class WhenIUseMyCustomHamcrestMatchers {
    @Test
    public void thehasSizeMatcherShouldMatchACollectionWithExpectedSize() {
        List<String> items = new ArrayList<String>();
        items.add("java");
        assertThat(items, hasSize(1));
    }
}
```

We want something like this...

I want a matcher that checks the size of a collection

Home-made Hamcrest Matchers

- Writing your own matchers in three easy steps!

```
public class HasSizeMatcher extends TypeSafeMatcher<Collection<? extends Object>> {
    private Matcher<Integer> matcher;

    public HasSizeMatcher(Matcher<Integer> matcher) {
        this.matcher = matcher;
    }

    public boolean matchesSafely(Collection<? extends Object> collection) {
       return matcher.matches(collection.size());
    }

    public void describeTo(Description description) {
        description.appendText("a collection with a size that is");
        matcher.describeTo(description);
    }
}
```

Extend the TypeSafeMatcher class

Provide expected values in the constructor

Do the actual matching

Describe our expectations

So let's write this Matcher!

Home-made Hamcrest Matchers

- Writing your own matchers in three easy steps!

```
import java.util.Collection;
import org.hamcrest.Factory;
import org.hamcrest.Matcher;

public class MyMatchers {

    @Factory
    public static Matcher<Collection<? extends Object>> hasSize(Matcher<Integer> matcher){
        return new HasSizeMatcher(matcher);
    }
}
```

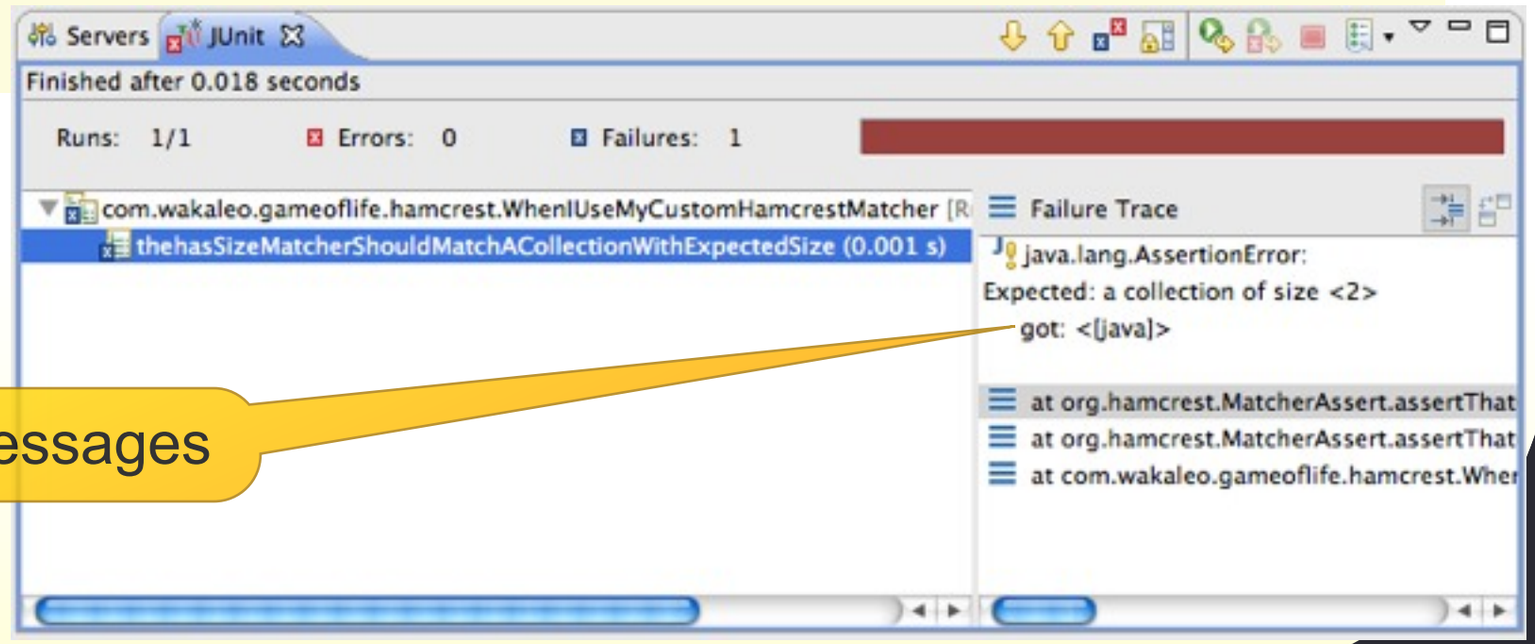
Use a factory class to store your matchers

All my custom matchers go in a special Factory class

Home-made Hamcrest Matchers

- Writing your own matchers in three easy steps!

```
import static com.wakaleo.gameoflife.hamcrest.MyMatchers.hasSize;
import static org.hamcrest.MatcherAssert.assertThat;

public class WhenIUseMyCustomHamcrestMatchers {

    @Test
    public void thehasSizeMatcherShouldMatchACollectionWithExpectedSize() {
        List<String> items = new ArrayList<String>();
        items.add("java");
        assertThat(items, hasSize(1));
    }
}
```



Hamcrest-style error messages

Home-made Hamcrest Matchers

- But wait! There's more!

```
@Test
public void weCanUseCustomMatchersWithOtherMatchers() {
    List<String> items = new ArrayList<String>();
    items.add("java");
    assertThat(items, allOf(hasSize(1), hasItem("java")));
}
```

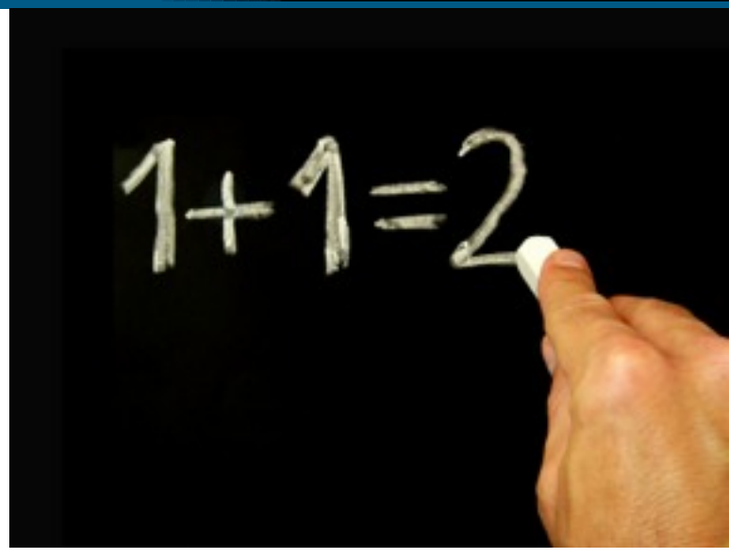
Combining matchers

```
@Test
public void weCanUseCustomMatchersWithOtherMatchers() {
    List<String> items = new ArrayList<String>();
    items.add("java");
    items.add("groovy");
    assertThat(items, hasSize(greaterThan(1)));
}
```

Nested matchers

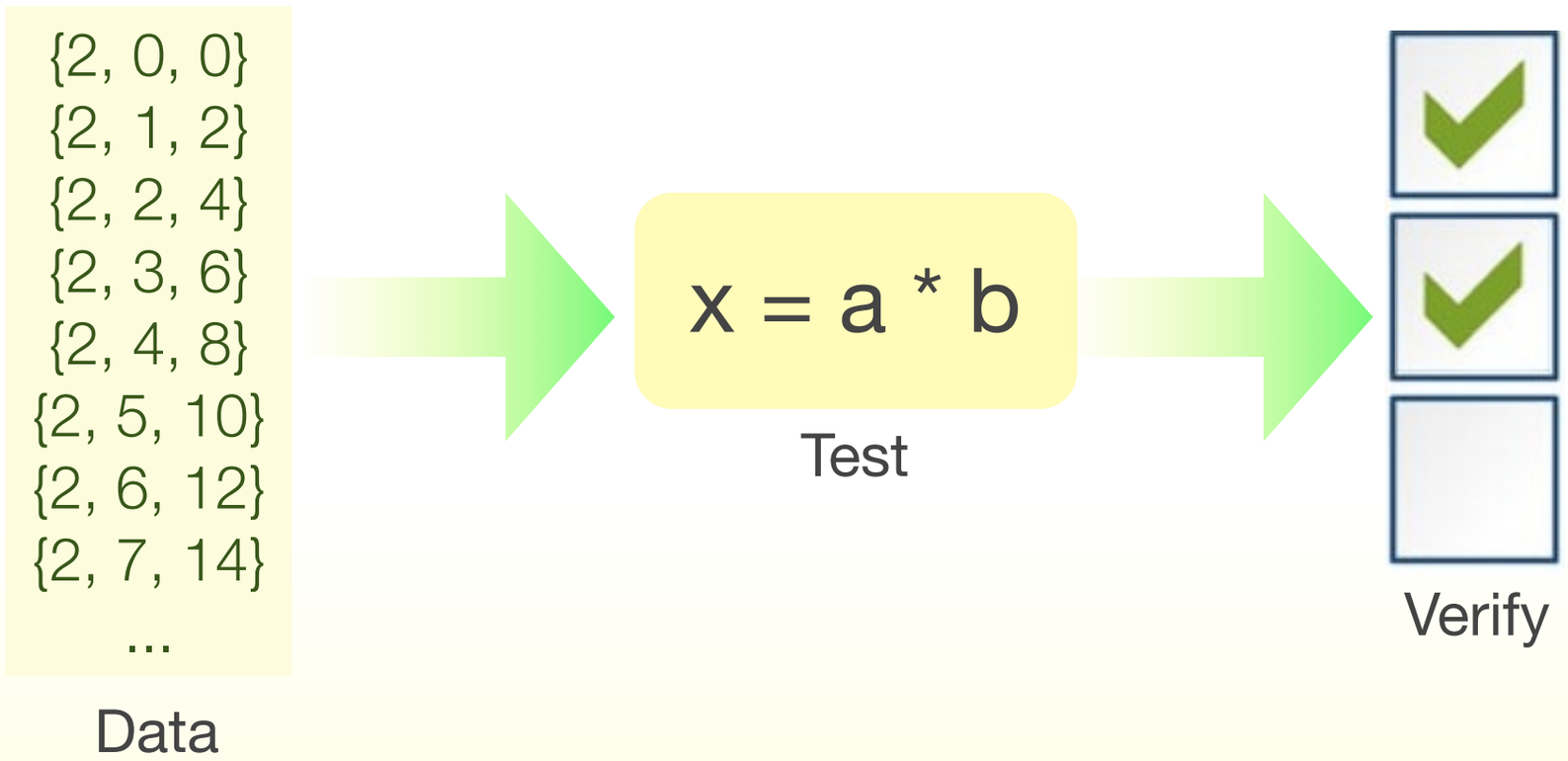
Data-Driven Unit Tests

- Using Parameterized Tests



Using Parameterized Tests

- Parameterized tests - for data-driven testing
  - Take a large set of test data, including an expected result
  - Define a test that uses the test data
  - Verify calculated result against expected result



Using Parameterized Tests

- Parameterized tests
  - Example: Calculating income tax

Taxable income	PAYE rate for every $1 of taxable income (excluding ACC earners' levy)
up to $14,000	12.5 cents
$14,001 to $48,000 inclusive	21 cents
$48,001 to $70,000	33 cents
$70,001 and over	38 cents

Using Parameterized Tests

- Parameterized tests with JUnit 4.8.1
  - What you need:
    - Some test data
    - A test class with matching fields
    - And some tests
    - And an annotation

```
@RunWith(Parameterized.class)
public class TaxCalculatorDataTest {
    private double income;
    private double expectedTax;

    public TaxCalculatorDataTest(double income, double expectedTax) {
        super();
        this.income = income;
        this.expectedTax = expectedTax;
    }

    @Test
    public void shouldCalculateCorrectTax() {...}
}
```

Income	Expected Tax
$0.00	$0.00
$10,000.00	$1,250.00
$14,000.00	$1,750.00
$14,001.00	$1,750.21
$45,000.00	$8,260.00
$48,000.00	$8,890.00
$48,001.00	$8,890.33
$65,238.00	$14,578.54
$70,000.00	$16,150.00
$70,001.00	$16,150.38
$80,000.00	$19,950.00
$100,000.00	$27,550.00

Using Parameterized Tests

How it works

```
@RunWith(Parameterized.class)
public class TaxCalculatorDataTest {
    private double income;
    private double expectedTax;

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0.00, 0.00 },
                { 10000.00, 1250.00 }, { 14000.00, 1750.00 },
                { 14001.00, 1750.21 }, { 45000.00, 8260.00 },
                { 48000.00, 8890.00 }, { 48001.00, 8890.33 },
                { 65238.00, 14578.54 }, { 70000.00, 16150.00 },
                { 70001.00, 16150.38 }, { 80000.00, 19950.00 },
                { 100000.00, 27550.00 }, });
    }

    public TaxCalculatorDataTest(double income, double expectedTax) {
        super();
        this.income = income;
        this.expectedTax = expectedTax;
    }

    @Test
    public void shouldCalculateCorrectTax() {
        TaxCalculator calculator = new TaxCalculator();
        double calculatedTax = calculator.calculateTax(income);
        assertThat(calculatedTax, is(expectedTax));
    }
}
```

This is a parameterized test

The @Parameters annotation indicates the test data

Income	Expected Tax
$0.00	$0.00
$10,000.00	$1,250.00
$14,000.00	$1,750.00
$14,001.00	$1,750.21
$45,000.00	$8,260.00
$48,000.00	$8,890.00
$48,001.00	$8,890.33
$65,238.00	$14,578.54
$70,000.00	$16,150.00
$70,001.00	$16,150.38
$80,000.00	$19,950.00
$100,000.00	$27,550.00

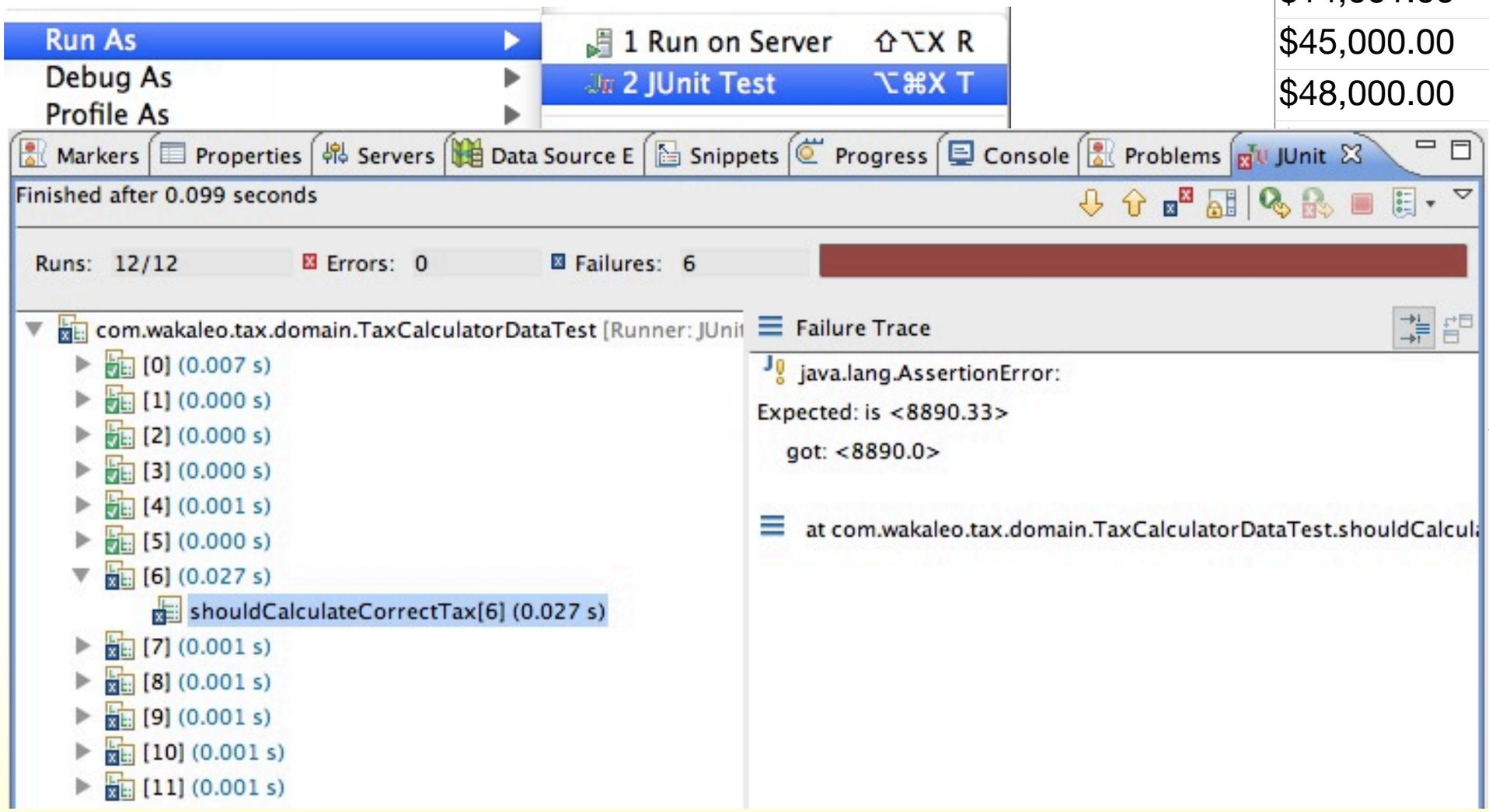
The constructor takes the fields from the test data

The unit tests use data from these fields.

Using Parameterized Tests

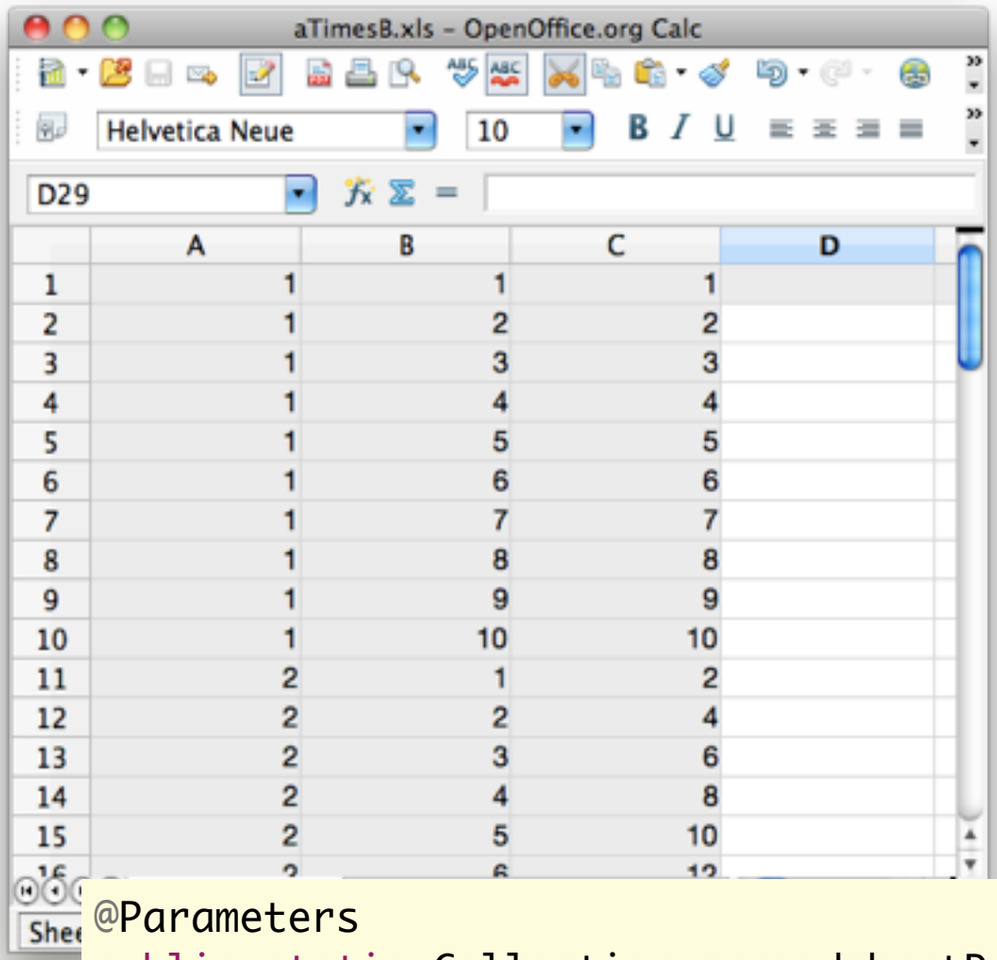
- Parameterized Tests in Eclipse
  - Run the test only once
  - Eclipse displays a result for each data set

Income	Expected Tax
$0.00	$0.00
$10,000.00	$1,250.00
$14,000.00	$1,750.00
$14,001.00	$1,750.21
$45,000.00	$8,260.00
$48,000.00	$8,890.00
	$8,890.33
	$14,578.54
	$16,150.00
	$16,150.38
	$19,950.00
	$27,550.00



Using Parameterized Tests

- Example: using an Excel Spreadsheet



	A	B	C	D
1	1	1	1	
2	1	2	2	
3	1	3	3	
4	1	4	4	
5	1	5	5	
6	1	6	6	
7	1	7	7	
8	1	8	8	
9	1	9	9	
10	1	10	10	
11	2	1	2	
12	2	2	4	
13	2	3	6	
14	2	4	8	
15	2	5	10	

```
@Parameters
public static Collection spreadsheetData() throws IOException {
    InputStream spreadsheet = new FileInputStream("src/test/resources/aTimesB.xls");
    return new SpreadsheetData(spreadsheet).getData();
}
```

Using Parameterized Tests

- Example: testing Selenium 2 Page Objects

A Page Object class

```
public class ExportCertificateSubmissionFormPage extends WebPage {

    @FindBy(name="HD_EN") WebElement importerName;
    @FindBy(name="HD_EA") WebElement importerAddress;
    @FindBy(name="HD_ER") WebElement importerRepresentative;
      ...
    public String getImporterName() {
        return importerName.getValue();
    }

    public void setImporterName(String value) {
        enter(value, into(importerName));
    }
      ...
}
```

Using Parameterized Tests

- Example: testing Selenium 2 Page Objects

Testing the fields using a parameterized test

```
@Parameters
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
            { "importerName", "an-importer-name" },
            { "importerAddress", "an-importer-address" },
            { "importerRepresentative", "a-importer-representative" },
              ...
            });
}
```

Using Parameterized Tests

- Example: testing Selenium 2 Page Objects

Setting up the test data

```
private String field;
private String inputValue;

public ReadingAndWritingFieldsWhenRaisingACertificate(String field,
                                                      String inputValue) {
    this.field = field;
    this.inputValue = inputValue;
}
```

Using Parameterized Tests

- Example: testing Selenium 2 Page Objects

Test reading and writing to/from the field

```
@Test
public void userShouldBeAbleToWriteToAnInputFieldAndThenReadTheValue() {
    setFieldValueUsing(setterFor(field));

    String retrievedValue = getFieldValueUsing(getterFor(field));

    assertThat(retrievedValue, is(inputValue));
}
```

(Reflection magic goes here)

JUnit Rules

- Using Existing and Custom JUnit Rules
  - Customize and control how JUnit behaves

JUnit Rules

The Temporary Folder Rule

```
public class LoadDynamicPropertiesTest {

    @Rule
    public TemporaryFolder folder = new TemporaryFolder();

    private File properties;

    @Before
    public void createTestData() throws IOException {
        properties = folder.newFile("messages.properties");
        BufferedWriter out = new BufferedWriter(new FileWriter(properties));
        // Set up the temporary file
        out.close();
    }

    @Test
    public void shouldLoadFromPropertiesFile() throws IOException {
       DynamicMessagesBundle bundle = new DynamicMessagesBundle();
       bundle.load(properties);
       // Do stuff with the temporary file
    }
}
```

Create a temporary folder

Prepare some test data

Use this folder in the tests

The folder will be deleted afterwards

JUnit Rules

- The ErrorCollector Rule
  - Report on multiple error conditions in a single test

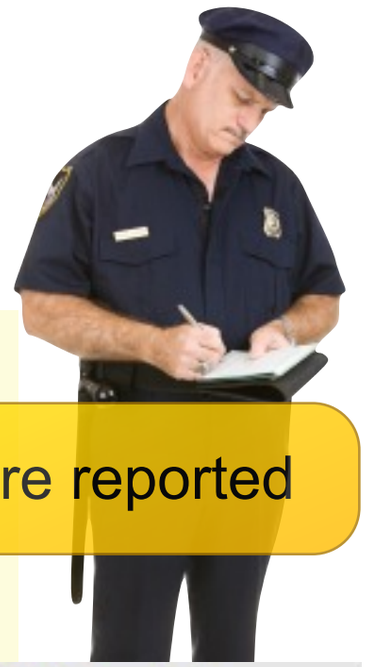
```
public class ErrorCollectorTest {

    @Rule
    public ErrorCollector collector = new ErrorCollector();

    @Test
    public void testSomething() {
        collector.addError(new Throwable("first thing went wrong"));
        collector.addError(new Throwable("second thing went wrong"));
        String result = doStuff();
        collector.checkThat(result, not(containsString("Oh no, not again")));
    }

    private String doStuff() {
        return "Oh no, not again";
    }
}
```

Two things went wrong here

Check using Hamcrest matchers

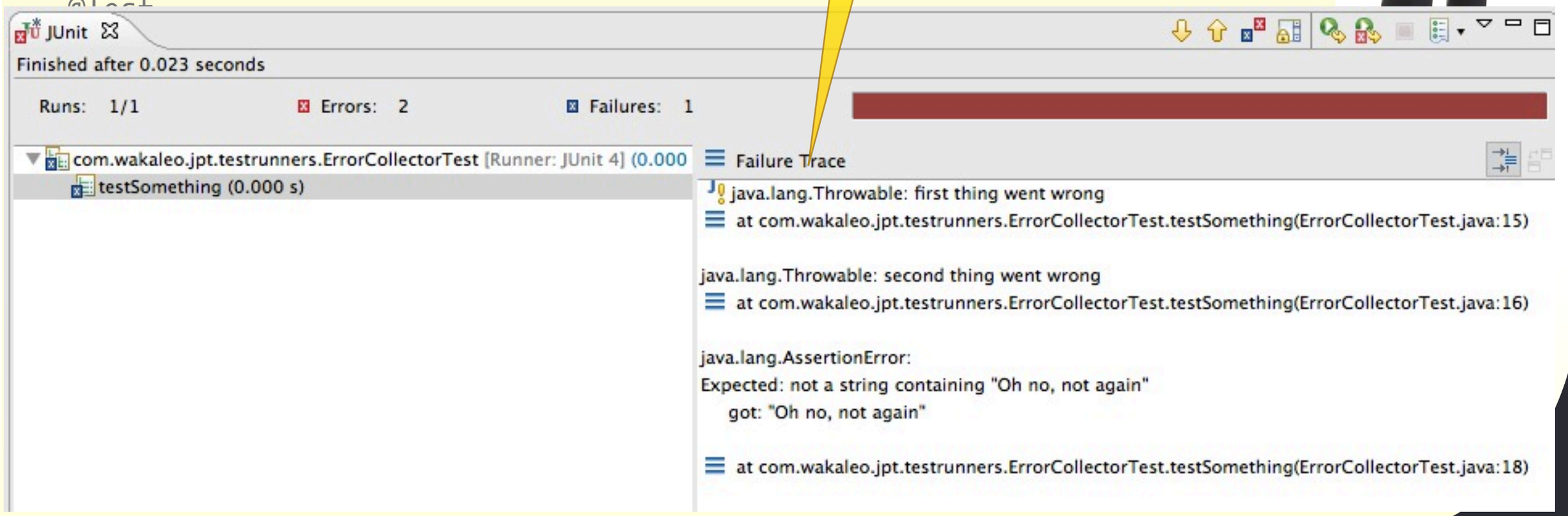
JUnit Rules

- The ErrorCollector Rule
  - Report on multiple error conditions in a single test



```
public class ErrorCollectorTest {

    @Rule
    public ErrorCollector collector = new ErrorCollector();

    @Test
```

All three error messages are reported



JUnit Rules

- The Timeout Rule
  - Define a timeout for all tests

```
public class GlobalTimeoutTest {

    @Rule
    public MethodRule globalTimeout = new Timeout(1000);

    @Test
    public void testSomething() {
        for(;;);
    }

    @Test
    public void testSomethingElse() {
    }
}
```

No test should take longer than 1 second

Oops

JUnit Rules

- The Verifier Rule
  - Adding your own custom verification logic

```
public class VerifierTest {

    private List<String> systemErrorMessages = new ArrayList<String>();

    @Rule
    public MethodRule verifier = new Verifier() {
        @Override
        public void verify() {
            assertThat(systemErrorMessages.size(), is(0));
        }
    };

    @Test
    public void testSomething() {
        // ...
        systemErrorMessages.add("Oh, bother!");
    }
}
```

After each method, perform this check

JUnit Rules

The Watchman Rule

Keeping an eye on the tests

```
public class WatchmanTest {
    private static String watchedLog;

    @Rule
    public MethodRule watchman = new TestWatchman() {
        @Override
        public void failed(Throwable e, FrameworkMethod method) {
            watchedLog += method.getName() + " " + e.getClass().getSimpleName() + "\n";
        }

        @Override
        public void succeeded(FrameworkMethod method) {
            watchedLog += method.getName() + " " + "success!\n";
        }
    };

    @Test
    public void fails() {
        fail();
    }
    @Test
    public void succeeds() {...}
}
```

Called whenever a test fails

Called whenever a test succeeds

JUnit Categories

- Grouping tests
  - Distinguish different types of tests
  - Mark tests using an annotation
  - Run different categories of test in different test suites

JUnit Categories

- Setting up JUnit Categories
  - 1) Define some categories

```
public interface IntegrationTests {}
```

Categories are defined as interfaces

```
public interface PerformanceTests {}
```

```
public interface PerformanceTests extends IntegrationTests {}
```

Subclassing works too

JUnit Categories

- Setting up JUnit Categories
  - 2) Annotate your tests

```
public class CellTest {

    @Category(PerformanceTests.class)
    @Test
    public void aLivingCellShouldPrintAsAPlus() {...}

    @Category(IntegrationTests.class)
    @Test
    public void aDeadCellShouldPrintAsAMinus() {...}

    @Category({PerformanceTests.class,IntegrationTests.class})
    @Test
    public void thePlusSymbolShouldProduceALivingCell() {...}

    @Test
    public void theMinusSymbolShouldProduceADeadCell() {...}
}
```

A test with a category

A test with several categories

A normal test

JUnit Categories

- Setting up JUnit Categories
  - 2) Or annotate your class

```
@Category(IntegrationTests.class)
public class CellTest {

    @Test
    public void aLivingCellShouldPrintAsAPlus() {...}

    @Test
    public void aDeadCellShouldPrintAsAMinus() {...}

    @Test
    public void thePlusSymbolShouldProduceALivingCell() {...}

    @Test
    public void theMinusSymbolShouldProduceADeadCell() {...}
}
```

You can also annotate the class

JUnit Categories

- Setting up JUnit Categories
  - 3) Set up a test suite

```
import org.junit.experimental.categories.Categories;
import org.junit.experimental.categories.Categories.IncludeCategory;
import org.junit.runner.RunWith;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Categories.class)
@IncludeCategory(PerformanceTests.class)
@SuiteClasses( { CellTest.class, WhenYouCreateANewUniverse.class })
public class PerformanceTestSuite {

}
```

Run only performance tests

Still need to mention the test classes

JUnit Categories

- Setting up JUnit Categories
  - 3) Set up a test suite - excluding categories

```
import org.junit.experimental.categories.Categories;
import org.junit.experimental.categories.Categories.IncludeCategory;
import org.junit.runner.RunWith;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Categories.class)
@ExcludeCategory(PerformanceTests.class)
@SuiteClasses( { CellTest.class, WhenYouCreateANewUniverse.class })
public class UnitTestSuite {

}
```

Don't run performance tests

Parallel tests

- Running tests in parallel
  - Run your tests faster
  - Also works well with multi-core CPUs
  - Results vary depending on the tests
    - IO
    - Database
    - ...

Parallel tests

- Setting up parallel tests with JUnit and Maven

```
<project...>
    <plugins>
      ...
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.5</version>
        <configuration>
          <parallel>methods</parallel>
        </configuration>
      </plugin>
    </plugins>
    ...
    <build>
      <dependencies>
       <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.8.1</version>
          <scope>test</scope>
        </dependency>
      </dependencies>
    </build>
  ...
</project>
```

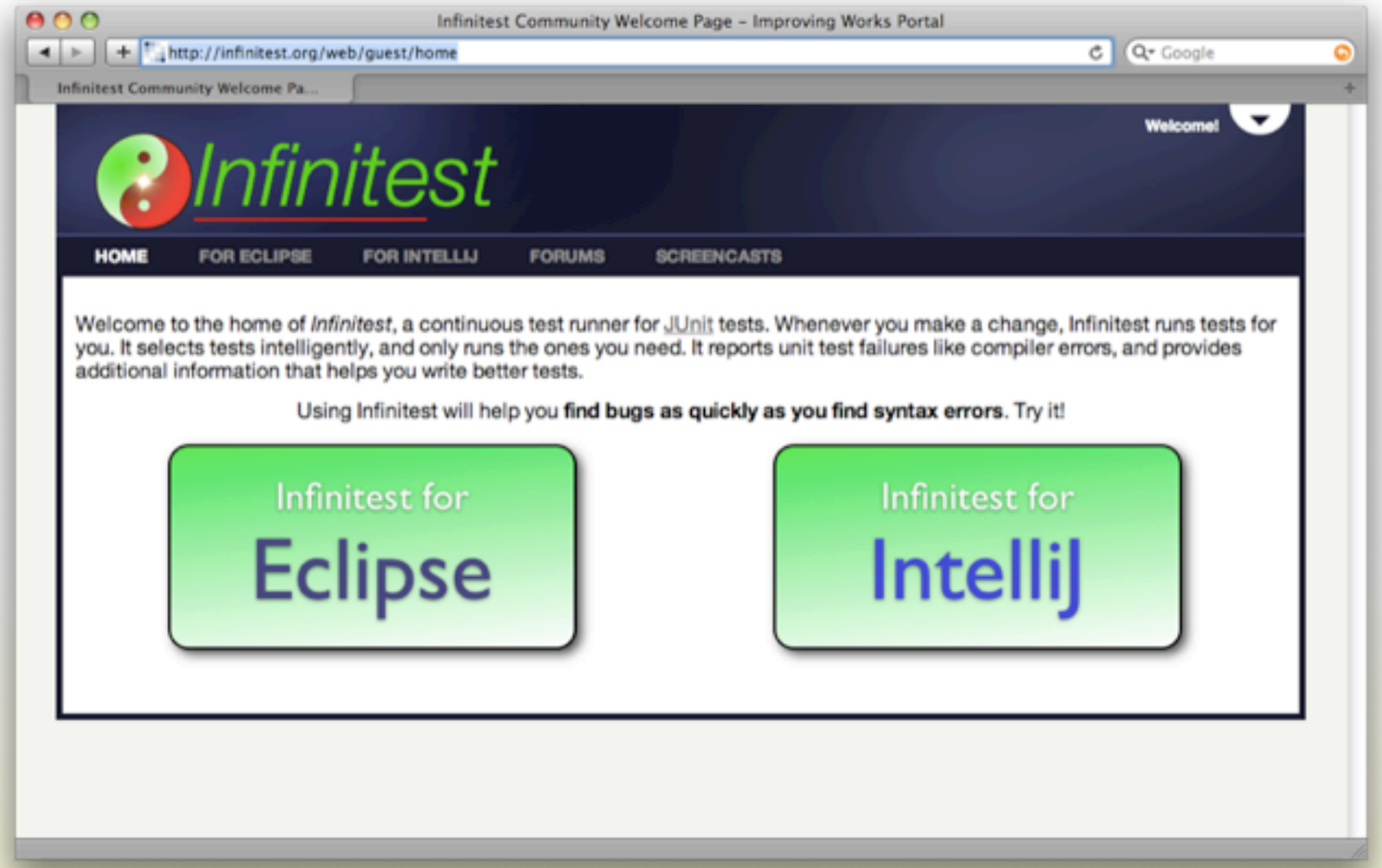
Needs Surefire 2.5

'methods', 'classes', or 'both'

Needs JUnit 4.8.1 or better

Continuous Testing

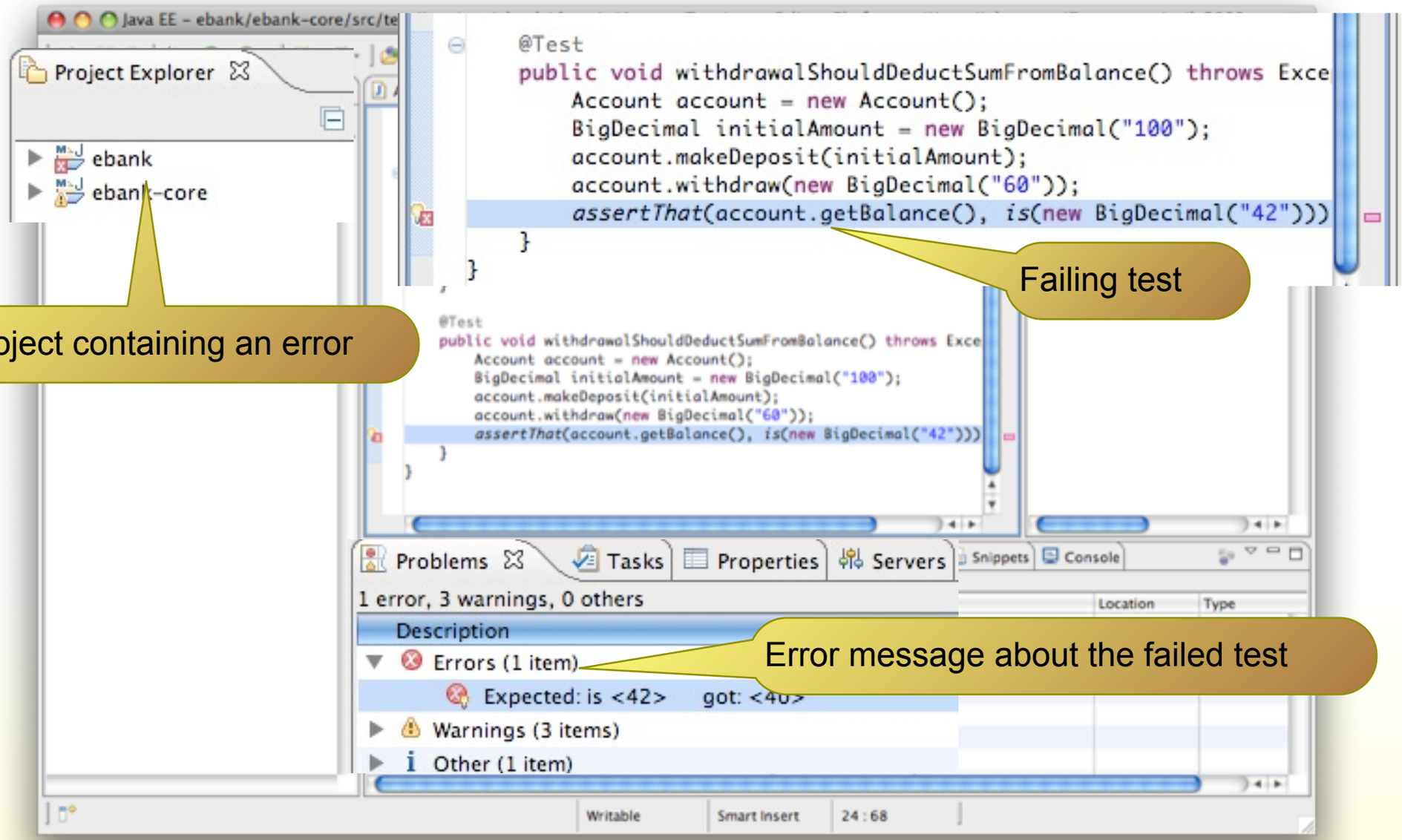
- Continuous Tests with Infinitest
  - Infinitest is a continuous test tool for Eclipse and IntelliJ
  - Runs your tests in the background when you save your code



Continuous Testing

Infinitest

- Using Infinitest
  - Whenever you save your file changes, unit tests will be rerun



Mocking with style

- Mockito - lightweight Java mocking

Account
balance number getFees(feeType)

```
import static org.mockito.Mockito.*;
....
Account accountStub = mock(Account.class);
when(accountStub.getFees(FeeType.ACCOUNT_MAINTENANCE)).thenReturn(4.00);
when(accountStub.getFees(FeeType.TRANSACTION_FEE)).thenReturn(0.50);

assertThat(accountStub.getFees(FeeType.TRANSACTION_FEE), is(0.50));
```

Low-formality mocking

Mocking with style

- Mockito - lightweight Java mocking

AccountDao
createNewAccount(String id)

```
AccountDao accountDao = mock(AccountDao.class);
Account newAccount = mock(Account.class);
when(accountDao.createNewAccount(123456)).thenReturn(newAccount)
                                          .thenThrow(new AccountExistsException() );
```

Manage successive calls

Mocking with style

- Mockito - lightweight Java mocking

AccountDao
createNewAccount(String id)

```
@Mock private AccountDao accountDao
...
```

Mockito annotations

Mocking with style

- Mockito - lightweight Java mocking

Account
balance number getEarnedInterest(year)

```
when(accountStub.getEarnedInterest(intThat(greaterThan(2000)))).thenReturn(10.00);
```

Use matchers

Mocking with style

- Mockito - lightweight Java mocking

AccountDao
createNewAccount(String id)

```
@Mock private AccountDao accountDao
...
// Test stuff
...
verify(accountDao).createNewAccount( (String) isNotNull());
```

Verify interactions