What's New in Java Land - 2

- junit (updated)
- mockito (new)

JUnit 4.4 - assertThat

```
assertThat(x, is(3));
assertThat(x, is(not(4)));

assertThat(
    responseString,

    either(containsString("color")).or(containsStrin
    g("colour")));

assertThat(myList, hasItem(3));
```

Junit 4.4 – assertThat

- More readable and typeable: this syntax allows you to think in terms of subject, verb, object (assert "x is 3") rather than assertEquals, which uses verb, object, subject (assert "equals 3 x")
- Combinations: any matcher statement s can be negated (not(s)), combined (either(s).or(t)), mapped to a collection (each(s)), or used in custom combinations (afterFiveSeconds(s))
- Custom Matchers. By implementing the Matcher interface yourself, you can get all of the above benefits for your own custom assertions.

Junit 4.4 – Assumptions

```
import static org.junit.Assume.*

@Test public void filenameIncludesUsername() {
   assumeThat(File.separatorChar, is('/'));
   assertThat(new User("optimus").configFileName(), is("configfiles/optimus.cfg"));
}

@Test public void correctBehaviorWhenFilenameIsNull() {
   assumeTrue(bugFixed("13356"));  // bugFixed is not included in JUnit
   assertThat(parse(null), is(new NullDocument()));
}
```

JUnit 4.4 – Assumptions

- With this release, a failed assumption will lead to the test being marked as passing, regardless of what the code below the assumption may assert. In the future, this may change, and a failed assumption may lead to the test being ignored: however, third-party runners do not currently allow this option.
- We have included assumeTrue for convenience, but thanks to the inclusion of Hamcrest, we do not need to create assumeEquals, assumeSame, and other analogues to the assert* methods. All of those functionalities are subsumed in assumeThat, with the appropriate matcher.
- A failing assumption in a @Before or @BeforeClass method will have the same effect as a failing assumption in each @Test method of the class.

JUnit 4.4 – Theories

```
@RunWith(Theories.class)
public class UserTest {
  @DataPoint public static String GOOD_USERNAME = "optimus";
  @DataPoint public static String USERNAME_WITH_SLASH = "optimus/prime";

  @Theory public void filenameIncludesUsername(String username) {
    assumeThat(username, not(containsString("/")));
    assertThat(new User(username).configFileName(), containsString(username));
  }
}
```

JUnit 4.4 – Theories

- More flexible and expressive assertions, combined with the ability to state assumptions clearly, lead to a new kind of statement of intent, which we call a "Theory". A test captures the intended behavior in one particular scenario. A theory captures some aspect of the intended behavior in possibly infinite numbers of potential scenarios.
- This makes it clear that the user's filename should be included in the config file name, only if it doesn't contain a slash. Another test or theory might define what happens when a username does contain a slash.
- UserTest will attempt to run filenameIncludesUsername on every compatible DataPoint defined in the class. If any of the assumptions fail, the data point is silently ignored. If all of the assumptions pass, but an assertion fails, the test fails.

JUnit 4.7 – Rules

```
public static class HasTempFolder {
@Rule
public TemporaryFolder folder= new TemporaryFolder();

@Test
public void testUsingTempFolder() throws IOException {
File createdFile= folder.newFile("myfile.txt");
File createdFolder= folder.newFolder("subfolder");
// ...
}
}
```

JUnit 4.7 – Rules

- Rules allow very flexible addition or redefinition of the behavior of each test method in a test class. Testers can reuse or extend one of the provided Rules, or write their own

- Rules are, in essence, another extension mechanism for JUnit, which can be used to add functionality to JUnit on a per-test basis. Most examples of custom runners in earlier versions of JUnit can be replaced by Rules.

JUnit - References

- Matchers, Assumptions and Theories

http://junit.sourceforge.net/doc/ReleaseNotes4.4.html

- Rules

http://github.com/KentBeck/junit/blob/d2da6a55bca4582c9a469f568df472a00e90ddf4/doc/ReleaseNotes4.7.txt

Mockito – why ?

- **Focused testing.** Should let me focus test methods on specific behavior/interaction. Should minimize distractions like expecting/recording irrelevant interactions.
- **Separation of stubbing and verification.** Should let me code in line with intuition: stub before execution, selectively verify interactions afterwards. I don't want any verification-related code before execution.
- **Explicit language.** Verification and stubbing code should be easy to discern, should distinguish from ordinary method calls / from any supporting code / assertions.
- **Simple stubbing model.** Should bring the simplicity of good old hand crafted stubs without the burden of actually implementing a class. Rather than verifying stubs I want to focus on testing if stubbed value is used correctly.

Mockito – example

Verifiying some behaviour

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);

//using mock object
mockedList.add("one");
mockedList.clear();

//verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

Stubbing

```
LinkedList mockedList =
    mock(LinkedList.class);

//stubbing
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new
    RuntimeException());

//following prints "first"
System.out.println(mockedList.get(0));

//following throws runtime exception
System.out.println(mockedList.get(1));
```

Mockito – features

- Mocks concrete classes as well as interfaces
- Little annotation syntax sugar - @Mock
- Verification errors are clean - click on stack trace to see failed verification in test; click on exception's cause to navigate to actual interaction in code. Stack trace is always clean.
- Allows flexible verification in order (e.g: verify in order what you want, not every single interaction)
- Supports exact-number-of-times and at-least-once verification
- Flexible verification or stubbing using argument matchers (anyObject(), anyString() or refEq() for reflection-based equality matching)
- Allows creating custom argument matchers or using existing hamcrest matchers

mockito – more examples

let's go to the http://mockito.org/

mockito - references

- Led by Szczepan Faber (used to work for ThoughtWorks)
- http://mockito.org/

koniec