TDD:
JUnit Introduction

Topics

- What is JUnit?
- JUnit annotations
  - @Test
  - @Before, @After
  - @BeforeClass, @AfterClass
- Assert statements
- AssertThat
- Testing tools
- Exception handling
- Parameterization
- Best practices

What is JUnit?

What is JUnit?

- Unit testing framework for Java
- De-facto standard
- Free and open-sourced

JUnit Annotations

@Test

@Before, @After

@BeforeClass, @AfterClass

@Ignore

How to write JUnit test?

- Create test class with test methods
- Each test method has method signature
  - Annotate it with @*Test*
  - public
  - void return type
  - No arguments
- Add fixtures
  - @*Before* and @*After* to run before/after each test method
  - @*BeforeClass* and @*AfterClass* to run once before/after all test methods
  - @*Ignore* to tempraily ignore the testing

Example Test class

```
public class CalculatorTest {
    private Calculator calculator;

   @Before
   public void setup() {
       calculator = new Calculator();
    }

    @Test public void
    should_return_addition_when_two_numbers_are_given() {
         assertEquals(5, calculator.add(2, 3));
         assertThat(calculator.add(2, 3), is(5));
    }

    @Test public void
    should_return_subtraction_when_two_numbers_are_given() {
          assertEquals(-1, calculator.subtract(2, 3));
          assertThat(calculator.subtract(2, 3), is(-1));
    }
}
```

Assert Statements

Assert Statements

- JUnit Assertions are methods starting with *assert*
- Determines the success or failure of a test
- An assert is simply a comparison between an expected value and an actual value
- Two variants
  - assertXXX(...)
  - assertXXX(String message, ...) - the message is displayed when the assertXXX() fails

Assert Statements

- Asserts expected.equals(actual) behavior
  - assertEquals(expected, actual)
  - assertEquals(String message, expected, actual)
- Asserts expected == actual behavior
  - assertSame(Object expected, Object actual)
  - assertSame(String message, Object expected, Object actual)
- Asserts that a condition is true
  - assertTrue(boolean condition)
  - assertTrue(String message, boolean condition)
- Asserts that a condition is false
  - assertFalse(boolean condition)
  - assertFalse(String message, boolean condition)

Assert Statements

- Asserts object reference is null
  - assertNull(Object obj)
  - assertNull(String message, Object obj)
- Asserts object reference is not null
  - assertNotNull(Object obj)
  - assertNotNull(String message, Object obj)
- Forces a failure
  - fail()
  - fail(String message)

Hamcrest AssertThat

What is Hamcrest?

- Hamcrest is a framework for writing matcher objects, allowing 'match' rules to be defined declaratively
- These matchers are used extensively in JUnit
- JUnit 4.4 introduced a new assert "assertThat" using
  - *org.hamcrest.CoreMatchers*
  - *org.junit.matchers.JUnitMatchers*
- Usage: assertThat([value], [matcher statement]);

Example: Hamcrest assertThat

```
@Test
    public void testDriverOne(){
          assertThat(driver.getAge(), is(18));
          assertThat(driver.getAge(), is(not(17)));
          assertThat(driver.getName(), containsString(“Kyaw ”));
         assertThat(driver.getFirstName(),
                        either(containsString(“mun”)).or(containsString(‘abc’)));
}
```

Why Hamcrest?

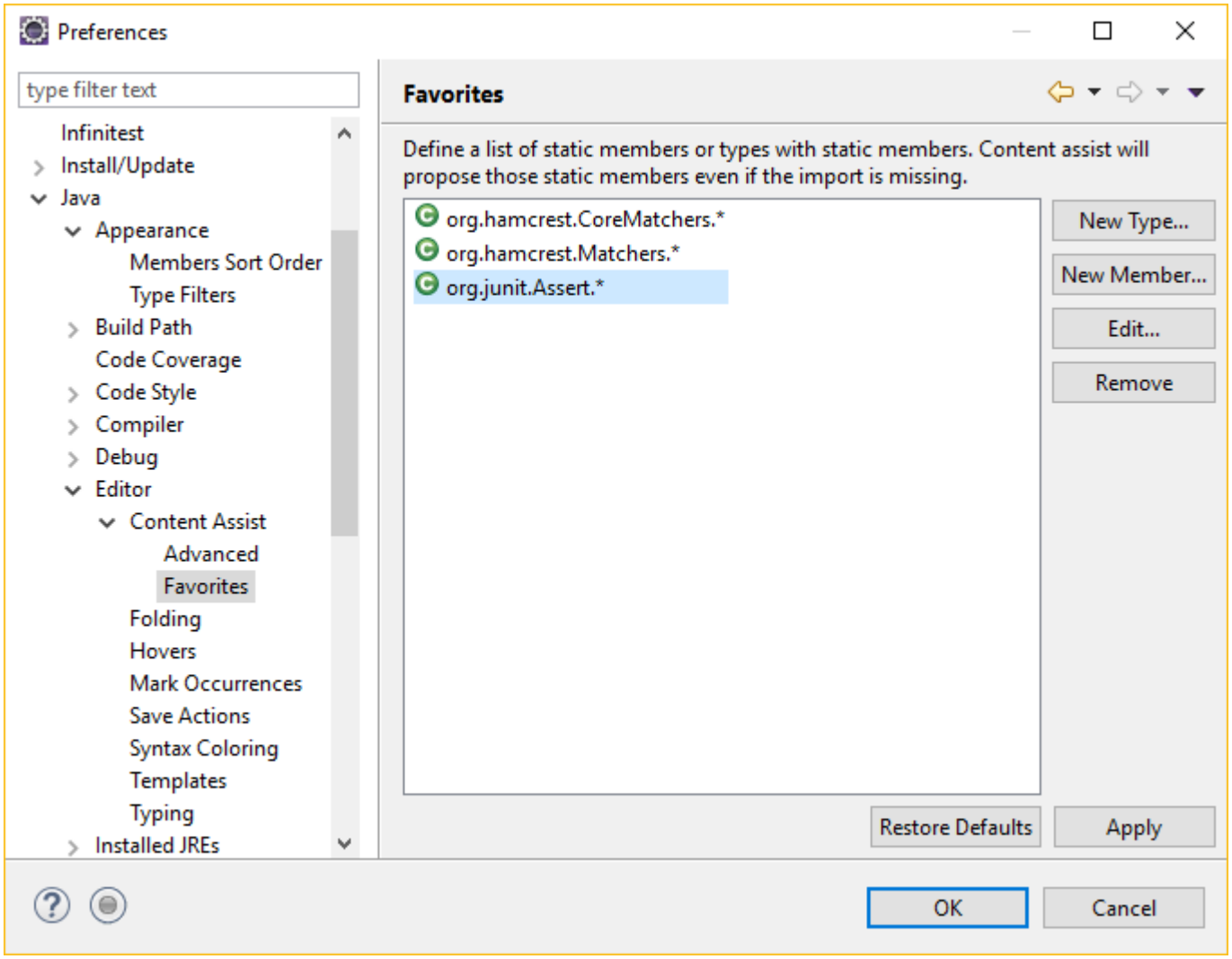
- More read'able and type'able
  - This syntax allows you to think in terms of subject, verb, object (assert “x is 3”) rather than assertEquals which uses verb, object, subject(assert “equals 3 x”)
- Combinations
  - Any matcher statements can be negated (not(s))
  - Any matcher statements can be combined (either(s) or (t))

Eclipse: Java->Editor>Content-Assist->Favorites

- Eclipse IDE cannot always create the corresponding static import statements automatically.
- You can configure the Eclipse IDE to use code completion to insert typical JUnit method calls and to add the static import automatically. For this open the Preferences via Window ▸ Preferences and select Java ▸ Editor ▸ Content Assist ▸ Favorites-> New Type

- *org.junit.Assert*
- *org.hamcrest.CoreMatchers*
- *org.hamcrest.Matchers*

Eclipse: Java->Editor>Content-Assist->Favorites



Lab:

Exercise 1: JUnit Annotations
1048_javase_junit.zip

Tools

Testing Tools

- Automatic testing tools
  - Each time a change is made on the source code, tests are run
  - Infinitest
- Code coverage tools
  - Used to describe the degree to which the source code of a program is executed when a particular test suite runs
  - EclEmma, JaCoco
- Source code version control
  - Git
- Continuous Build tools
  - Jenkins

Lab:

Exercise 2: Installation of Tools 1048_javase_junit.zip

Exception Handling

Ways of handling exceptions in JUnit

- @Test(expected = <Exception.class>)
- @Rule
- try-catch idiom

@Test(expected = <Exception.class>)

- Supported in JUnit 4

```
@Test (expected = IllegalArgumentException.class)
public void throwsExceptionWhenNegativeNumbersAreGiven() {
    calculator.add("-1,-2,3"); // This throws an IllegalArgumentException
}
```

@Rule

- Supported in JUnit 4
- The rule must be a public field marked with @Rule annotation

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void throwsExceptionWhenNegativeNumbersAreGiven() {
    // arrange
    thrown.expect(IllegalArgumentException.class);
    thrown.expectMessage(equalTo("negatives not allowed: [-1, -2]"));
    // act
    calculator.add("-1,-2,3");
}
```

try-catch

- Supported in JUnit 3 and JUnit 4
- The test will fail when no exception is thrown and the exception itself is verified in a catch clause
- Least favored option

```
@Test
public void throwsExceptionWhenNegativeNumbersAreGiven() {
    try {
        calculator.add("-1,-2,3");
        fail("Should throw an exception if one or more of given numbers are negative");
    } catch (Exception e) {
        assertThat(e)
                .isInstanceOf(IllegalArgumentException.class)
                .hasMessage("negatives not allowed: [-1, -2]");
    }
}
```

Lab:

Exercise 3: Exception Handling

1048_javase_junit.zip

Parameterization

Parameterized Testing

- The Parameterized Test is used when we find ourselves writing identical tests where only a few data input values are different but the logic is the same
- Only one test method is needed which will have parameterized data supplied to it

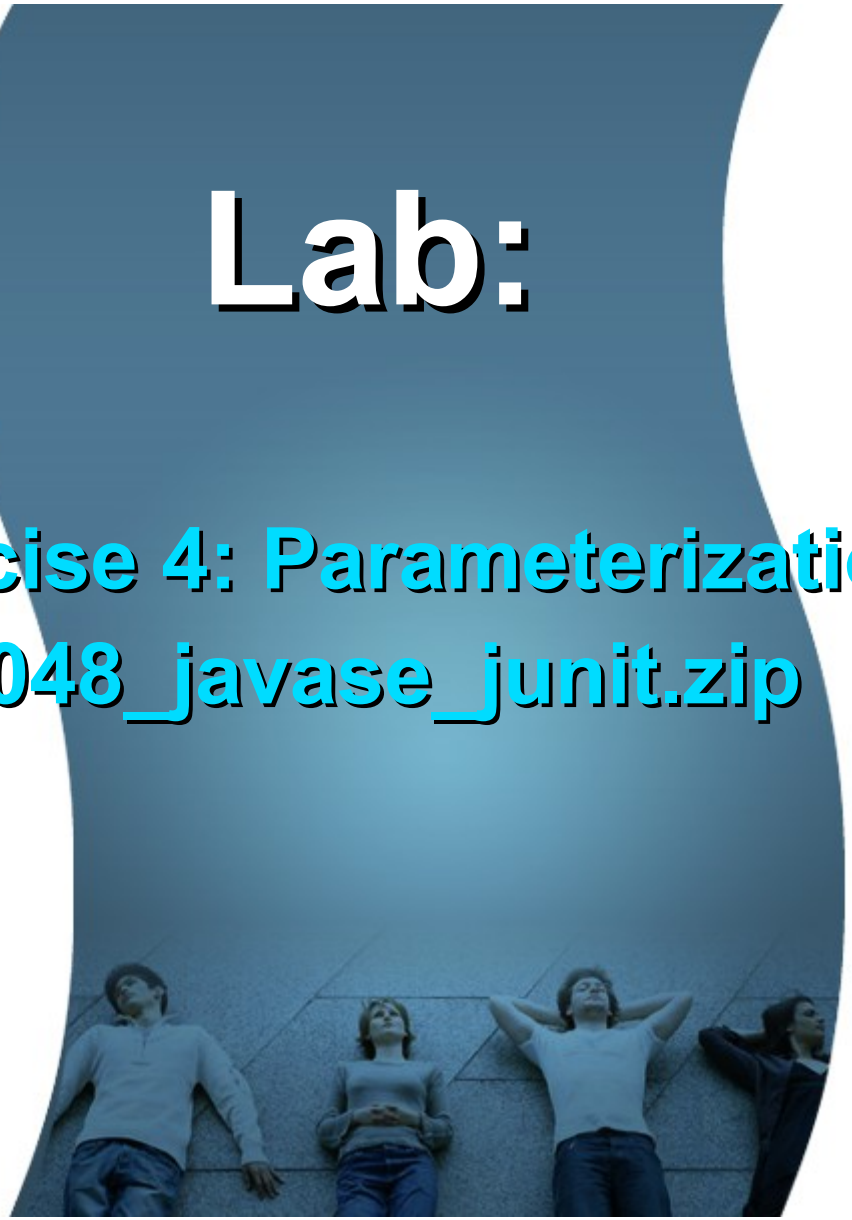
Example: Parameterized testing

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

    private Calculator calculator;
    private int x;
    private int y;
    private int result;

    public CalculatorTest(int x, int y, int result) {
        this.x = x;
        this.y = y;
        this.result = result;
    }

    @Parameters
    public static Collection testValues() {
    return Arrays.asList(new Object[][] {
        { 2, 3, 5 },
        { 4, 9, 13 },
        { 1, 9, 10 },
      });
}
```

Lab:

Exercise 4: Parameterization

1048_javase_junit.zip



JUnit Best Practices

Test Code vs Production Code

- Your test code becomes part of the codebase
- It is backed up and put into a source code repository along with the production code
- Test code needs to be documented, although not as heavily as the production code
- Note that test code will most likely not deploy with the production code

Be thorough

- Write tests for everything
  - Except methods that are "too simple to fail"
  - Only trivial getter/setter methods are too simple to fail
- Write multiple tests for each method
  - "positive" testing
  - "negative" testing
  - boundary value testing (BVT)

Thank You!