1. Introduction

1. Why am I doing this?
    1. My experience
    1. Helpful features
    1. **Y**ou decide if it is suitable
    1. **no** syntactical sugar

1. The point of this presentaion
    1. **presentation.PresentationTest**
    1. start really basic

1. Hamcreast matchers
    1. **a.BikeTest**
        1. **Bike** class looks like this
        1. assertTrue
            1. explicit argument
            1. RUN *firstBike*
            1. fail messages
        1. assertThat
            1. no arguments
            1. compares data types
            1. RUN *secondBike*
            1. better fail messages
            1. complex data structures

1. other types of matchers
    1. *firstMatchers*
    1. *is* uses *equals()*
    1. *contains*, content of a list in a specific order
    1. *hasItem*, specific item in a list without order
    1. we haven't reduced lines of code yet
1. combinable matchers
    1. *secondMatchers*
    1. *combinableMatchers*
    1. type safe, can't mix e.g. *hasItem* and *hasSize*
    1. builder pattern
    1. can be combined
    1. awkward for a large object
1. custom matcher
    1. *firstCustomMatchers*
    1. lets use the *is* matcher
    1. usually, but our **Bike** is a bit awkward, manufacturingDate
    1. lets create our own matcher
1. create custom matcher
    1. *secondCustomMatchers*
    1. sprouted a method
    1. **Macther** hierarchy

1. **TypeSafeMatcher** sounds promising
1. **CustomTypeSafeMatcher** the one we want
1. provide, description and an implementation

1. finished custom matcher
1. finished matcher can look like
1. usually, *toString* but that can be overridden

1. Mockito argument matchers
1. **a.BikeServiceTest**
1. using mocks you might run in to the same issue
1. not using argument matcher compared by reference
1. create our own argument matcher
1. similar to hamcreast matcher but a bit more
1. two ways of doing this
1. first way, register an argument matcher
1. second way, use argsThat

1. Parametrized tests
1. **a.BikeForTest**
1. we had this in the beginning
1. how to test for input and expected output mutations
1. one i tried early is loop
1. obviously horrible due to, which iteration failed
1. **a.BikeParametrizedTest**
1. the solution is called, parametrized test or data driven test
1. require a junit runner
1. in this case Parametrized
1. specifying data to test with
1. requires constructor and fields
1. object array input to the constructor
1. the collection is looped over
1. fields "pass" data to the test
1. for every instance all test cases run with instances data
1. can't mix in junit 4
1. others might allow this, jest
1. report isn't much better
1. name variable for the test

        1. don't over do it like **product - CreateBodyParametrizedTest**
    1. **b.PaintShopTest**
        1. hamcrest matcher in the data
        1. additions to enum
        1. don't need to bother about renaming test will break if we mess up
    1. **b.FancyColourTest**
        1. new test, will catch additions
        1. not catch simultaneous additions and deletions
        1. jpa annotation, by using reflection

1. Rules
    1. only used them a couple of times
    1. lots of set up to do before a test
    1. try using rules or class rules
    1. a rule can be used instead of *@before*, *@after*, *@beforeClass* and *@afterClass*
    1. **c.ExperimentExternalResourceTest**
        1. start with externalResource
        1. extend ExternalResource
        1. implement a couple of methods
        1. create instance of the rule
        1. annotate, either *@Rule* or *@ClassRule*
        1. run at bootstrapping
    1. **c.ExperimentMethodAndTestRuleTest**
        1. other rules, MethodRule and TestRule, only have and apply
        1. which is basically a *@before*
        1. other provided, TemporaryFolder and TestName

1. More JunitRunners
    1. explore an other junit runner
    1. **d.FancyServiceTest**
        1. annotate a variable to mock it
        1. runner will do mock=mock(FancyService.class)
        1. a bit clearer code
        1. use *@Mock* in a data driven test
    1. **d.FancyServiceParametrizedTest**
        1. alter the test a bit
        1. Parametrized runner for this
        1. of course we need our data
        1. *@Before* fixture, manually do what the Mockito runner does
    1. *SpringJUnit4ClassRunner* in combination with parametrized and/or mockito runner

        1. *SpringClassRule and SpringMethodRule* for that
        1. **product – GetSubCategoryIntegrationTest**

1. Conclusion
    1. we begun with this
    1. and now we have **ResultTest** and **ResultParametrizedTest**

1. Key take aways
    1. macthers can make your test easier to understand
    1. parametrized test can help a lot, but use them responsibly
    1. rules can be helpful with your setup
    1. the mockito runner helps you to focus on the important parts of mocking
    1. you can combine all of these features at the same time in a test, but please don't