Exercise 1 (JUnit - annotations mm.)

1. Skriv en lille klasse med et par simple metoder
   ex: Account med konstruktør, deposit(), withdraw(), getBalance()

2. @Test
   Skriv et par testmetoder, der indeholder de 4 elementer (setup, exercise, expectation, validation)
   Kør: hø klik test-fil > Run File

3. Udvid projektet med endnu en klasse med en metode
   Skriv en lille test af klassen
   Kør begge test: Run > Test Project

4. @Before
   Saml fixtur-kode i setUp() (testen af Account)

5. Synlig intention.
   Skriv en testmetode, der tester, at overtræk ikke er muligt. Lad intentionen fremgå af metodens navn.

6. @Test(expected = <navn på Exception>.class)
   Afprøv

7. @Test(timeout = <tid i ms>)
   Afprøv

8. Suite
   Køre en speciel delmængde af alle testene i et projekt:
   Skab suiten: hø klik test-pakke > Test Suite >>> (medtager test i samme pakke)
   Kør: hø klik på suite-fil > Run File

   Redigér suiten, så
   - visse tests udelukkes
   - den inkluderer test fra andre pakker (end dén, den selv ligger i)

Mere:
https://netbeans.org/kb/docs/java/junit-intro.html#Exercise_42

Exercise 2 (Test First development)

Write a ”text formatter” that can take arbitrary strings and horizontally center them on a line.

Follow the steps in TFD. The test is always written first and compiled. If the compile fails, then production code is added to make the compile succeed. Then the test is run to see if it passes. If the test fails, then production code is added to make the test pass. If the test passes, then a new test is added.

Ex: Class: Formatter

Methods: setLineWidth(10)

center(”word”) (returns : ” word ”)

From

http://epf.eclipse.org/wikis/xp/xp/guidances/guidelines/test_driven_development_tdd_D7F1F440.html

Exercise 3 (Test First Development, Pair Programming)

String Calculator

The following is a TDD Kata- an exercise in coding, refactoring and test-first, that you should apply daily for at least 15 minutes (I do 30).

Before you start:

- Try not to read ahead.
- Do one task at a time. The trick is to learn to work incrementally.
- Make sure you only test for **correct inputs**. there is no need to test for invalid inputs for this kata

String Calculator

1. Create a simple String calculator with a method **int Add(string numbers)**
    1. The method can take 0, 1 or 2 numbers, and will return their sum (for an empty string it will return 0) for example **“” or “1” or “1,2”**
    2. Start with the simplest test case of an empty string and move to 1 and two numbers
    3. Remember to solve things as simply as possible so that you force yourself to write tests you did not think about
    4. Remember to refactor after each passing test
2. Allow the Add method to handle an unknown amount of numbers
3. Allow the Add method to handle new lines between numbers (instead of commas).
    1. the following input is ok: “1\n2,3” (will equal 6)
    2. the following input is NOT ok: “1,\n” (not need to prove it - just clarifying)
4.

    Support different delimiters

    1. to change a delimiter, the beginning of the string will contain a separate line that looks like this: “//[delimiter]\n[numbers…]” for example “//;\n1;2” should return three where the default delimiter is ‘;’ .
    2. the first line is optional. all existing scenarios should still be supported
5. Calling Add with a negative number will throw an exception “negatives not allowed” - and the negative that was passed.if there are multiple negatives, show all of them in the exception message

---

stop here if you are a beginner. Continue if you can finish the steps so far in less than 30 minutes.

---

6. Numbers bigger than 1000 should be ignored, so adding 2 + 1001 = 2
7. Delimiters can be of any length with the following format: “//[delimiter]\n” for example: “//[***]\n1***2***3” should return 6
8. Allow multiple delimiters like this: “//[delim1][delim2]\n” for example “//[*][%]\n1*2%3” should return 6.
9. make sure you can also handle multiple delimiters with length longer than one char

from
http://osherove.com/tdd-kata-1/

Exercise 4 (Refactoring)

Tag et kik på 5 tilfældigt valgte refaktoreringer fra Fowlers katalog: http://refactoring.com/catalog/

1) Læs og forstå, hvad de går ud på
2) Vælg dén refaktorering, der umiddelbart synes mest interessant at kunne huske ifm. TDD.
3) Beskriv den kort

Exercise 5 (stub, dependency injection, state based test)

Download and import the demo ”dep_inject_demo_publisher.zip”.

Make the following change to Publisher: When publish() is called, Publisher must collect the latest ”message content” from another ”source object” before it contacts the subscriber.

1. Declare a new interface
2. Make a stub class
3. Make/change the test!!
4. Make the changes to Publisher that make the test pass.

Exercise 6 (jMock – verify the framework on your computer)

1. Download and import the demo ”jMock_demo_Publisher.zip” (Fronter)
2. Add the following JAR files to the class path of the project (Properties > Libraries > Add JAR/Folder)
   a. jmock-2.6.0.jar
   b. hamcrest-core-1.3.jar
   c. hamcrest-library-1.3.jar
   d. jmock-junit4-2.6.0.jar
3. Run the test (should pass)

Exercise 7 (Behaviour Testing with Mock objects)

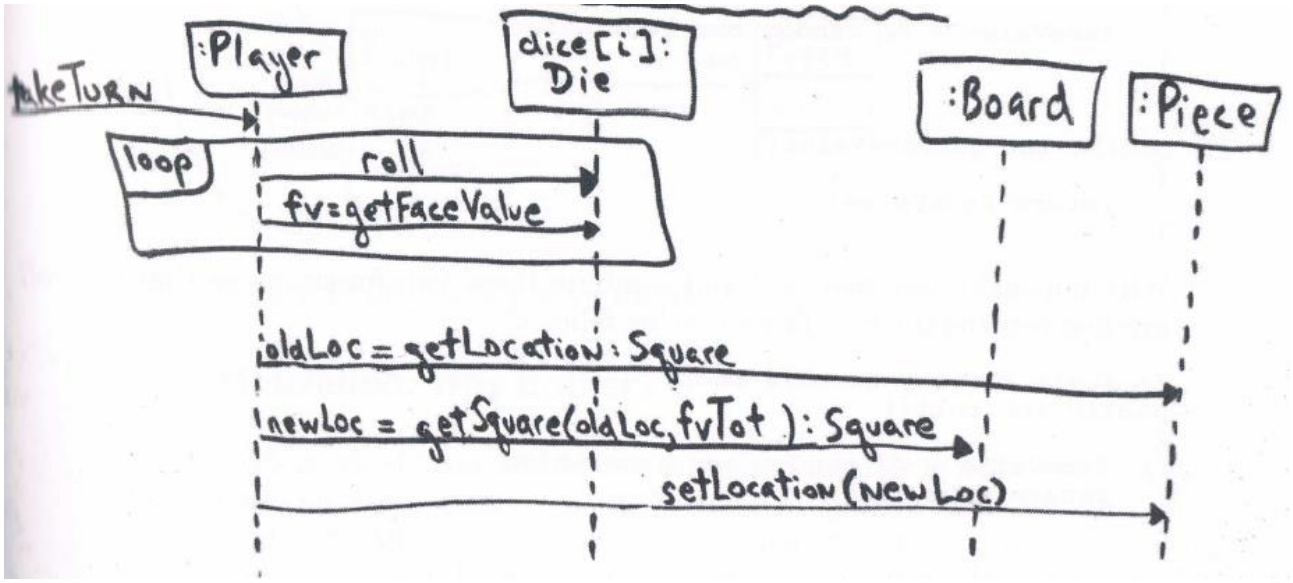
Use jMock to create a ”behaviour centered” test of the Player class in a Monopoly game.

Create mock of the objects that Player collaborate with and depends on.

Verify that the method takeTurn() behaves as expected. ”Take turn” means

1. generate two random numbers (the dice)
2. calculate new position
3. move piece to new position

UML sequence diagram illustrating the expected interactions.



Inspiration from
Larman: Applying UML and Patterns, 3. ed

Ressources:

http://jmock.org/index.html

- Download (latest version)
- Cheat Sheet (overview)
- Cookbook (how to ..)