# HÁSKÓLI ÍSLANDS
## Iðnaðarverkfræði-, vélaverkfræði- og tölvunarfræðideild

HBV205M: Prófun hugbúnaðar / Software Testing · Spring 2021
Dr. Helmut Neukirchen

**Assignment 10 · Due 15.3.2021, 10:00 – Submit sources of test classes**

*Objectives: Apply JUnit to create beautiful unit tests, experience encapsulation affecting testability, train test-driven development, train using mock objects by applying Mockito.*

In this assignment, we will create executable JUnit 4 tests for a Java class that implements a stack that stores `Money` objects and has a maximum capacity of 10 elements: first, tests are created in a traditional way (first implementation, then test), then in a test-driven development (TDD) / test-first style.

*Note:* Style matters for grading, e.g. using a fixture for objects that are common to test cases, having applied TDD by-the-book, covering states of the class under test, etc.

1. Download the Java implementation of the stack (`hbv205m_assignment10_MoneyStack.zip`) from Canvas.

   If you use Eclipse, you can import the zip file via: *Import → General → Existing Projects into Workspace → Next → Select Archive File → Browse... → Finish.*

   This gives you a project that has the JUnit 4 library, the complete Hamcrest 1.3 library ("hamcrest-all") if you want to use `assertThat`, and Mockito 3.x with all it's dependencies. The Eclipse project has Java 8 set as JDK version, but it should also work with newer versions.

2. Create JUnit 4 tests for the `push`, `pop`, and `isFull` methods of class `MoneyStack`. (In Eclipse, you can use *New → JUnit Test Case* to create some boilerplate code if you have selected `MoneyStack`. You can run the tests via *Run as → JUnit Test.*) It is OK to have all you code in the same default package as the implementation under test.

   (As the focus of this assignment is on JUnit, you do not need to apply formaly the FREE/N+ approach to create test cases for a state-based class, still you should take into account that a stack has states such as empty, full, and a state in-between.)

   Take care to create also negative test cases to test that the stack implementation throws `ArrayIndexOutOfBoundsException` when `push` and `pop` are used beyond the limits.

   *Notes:*

   - A correct stack should not simply store the passed amount of a `Money` object, but store the passed object itself. To check `pop` actually gives the same object back that was pushed, rather use `assertSame` – not `assertEquals`.
   - While creating tests, you will experience that the advantages of object-orientation (e.g. due to encapsulation, `push` and `pop` can only be tested together, i.e. `pop` must return what has been pushed before – instead of being able to test the internal data structures) are a disadvantage for testing.

3. Now, add in a Test-Driven Development style the methods `isEmpty` (returns a Boolean whether the stack is empty), `getCapacity` (returns the maximum allowed size of the stack as integer), and `getCurrentSize` (returns the current size of the stack as integer) to class `MoneyStack`.

4. Now that you have the method `getCapacity`, modify your test cases to use that method instead of a hardcoded capacity value.

5. Finally, create a new test class that is used for testing the `sum` method of the stack:

   (a) First, create a test case without any mock objects, i.e. using real `Money` objects.

   (b) Then, create another, additional test case that uses Mockito mock objects instead of any real `Money` objects that:

      i. checks that the stack implementation actually calls the `getAmount` method of `Money`

      ii. and that mocks that `getAmount` method to return to the stack those values that make sense for your test case.

   *Hint*: As you replace `Money` completely by a mock, there is no constructor with a parameter anymore (as `Money` would have) – you simply create a dumb mock that is so dumb that it does not even have a constructor. Any behaviour of the mock is therefore independent from any constructur parameter, but needs rather to be achieved by the other methods of the mock for `Money` that are replaced by mock methods.

6. Create a JUnit 4-style test suite to include all your test cases classes. (In Eclipse, simply use *New* → *Other* → *Java* → *JUnit* → *JUnit Test Suite*.)

As solution, only upload the final result of your source code containing the JUnit tests (and nothing else, e.g. do not upload byte code or whole Eclipse projects) – typically, these are just three `.java` files: one for the test suite, one with the test cases for method `sum`, and one with all the other test cases.

## To keep in mind during flipped classroom sessions

- Have audio working (use smartphone with Zoom app in case of problems, also non-audio problems. In case of crashes, download/install latest Zoom version).
- Switch on video to create a more personal atmosphere.
- First: Solution of last week's assignment 9 presented (Assignment 7 graded, but Assignment 8 not yet graded).
- Next: Questions concerning slides/videos (JUnit) discussed.
- Then: New assignment 10 (JUnit, Mockito) introduced.
- Finally: Work on assignments in breakout rooms: by default random allocation of 2 students each, but: If you want to work in the breakout rooms with your favourite buddy, this is possible as join can on your own move to breakout rooms. But it is recommended to let me know via the chat so the we can coordinate who is allocating which breakout room
  - Breakout rooms do not get recorded.
  - First thing to do:
    1. Remember breakout room number: in case you need to reconnect, Helmut can then add you again to your old breakout room.
       * **if Helmut remembers to enable, you can also join breakout rooms yourself** / if you join late, you can add yourself to an empty breakout room or one that has only one student.
    2. Exchange contact details (email/phone), e.g. via chat in each breakout room, so that you can continue in case of technical problems/after class finishes.
  - While there are Eclipse plug-ins for shared editing, I suggest to have one student editing and sharing the screen to the other student.
  - Use "Ask for help" button (question mark icon) to make Helmut join (but may take time if he is busy in another breakout room).
    * Redo "Ask for help" if Helmut does not come after a few minutes (the requests do not queue up, but Helmut sees them only displayed once). In case of being idle, Helmut will come along each breakout room.
  - Submit Java source code file to Gradescope as a team (unless you disagree on solution): use Gradescope's group submission feature. Gradescope is reachable via Canvas. You have until Monday for submission (re-submission possible).
  - In case of some announcement for all: Helmut sends a chat message to all or ends all breakout room sessions for a video session with all. (If Helmut does not mess it up, should be possible to continue in the old breakout room.)
- There is no wrap-up at end of class, i.e. if you are finished with your assignment, there is no need to wait – you are welcome to leave the class.
- Typically, there are no Wednesday classes: use the time to read new slide/watch videos as preperation for next Monday.
- Today or tomorrow, you get the remaining slides of Chapter 8: PIT mutation testing tool usage and Cucumber behaviour driven development. In addition, you will later this week get a new Chapter 9 on test tools in general: the remaining assignments will be on these topics.
- Depending on the situation, the exam might be changed from oral to written.