Debugging and Checkers Intro

Marcel Zauder

March 26, 2021

Exercise 04

Tasks

1. Test **Game#play(IDie)** with two different IDies: one mocked by hand, one mocked using Mockito
2. Compare these two approaches
3. Test all **Squares** in the game, use Mockito to mock unrelated objects
4. Add a new square: SpeedUpSquare, test it
5. Cover the code

Exercise 04

In **Game.java**:

```
public void play(Die die) {
    ...
}
```

Change to:

```
public void play(IDie die) {
    ...
}
```

Exercise 04

Then test with:

```
@BeforeEach
public void initializeTest() {
    ...
    testGame = new Game(GAMESIZE,players,DIESIDES);
    IDie mockDie = mock(IDie.class);
    when(mockDie.roll()).thenReturn(1, 2, 5, 4, ...);
    testGame.play(mockDie);
}
```

@BeforeEach is called before each test-run. @BeforeAll is called once when the wohle test is started.

Exercise 04

Another mocking example:

```
@Test
public void testPlayerSwapOnly(){
    Game mkGame = mock(Game.class);
    FirstSquare mkFirstSquare = mock(FirstSquare.class);
    LastSquare mkLastSquare = mock(LastSquare.class);
    when(mkGame.firstSquare()).thenReturn(mkFirstSquare);
    when(mkGame.getSquare(2)).thenReturn(mkLastSquare);
    when(mkLastSquare.position()).thenReturn(2);
    Player Jack = new Player("Jack");
    Jack.joinGame(mkGame);
    Jack.swap(mkLastSquare);
    assertEquals(2,Jack.position());
}
```

The *swap* behaviour is implemented in the **Player**, so we mock the **Game** and the **Squares**.

Exercise 04

Mocking Tips

1. Don't mock the object that you're trying to test - that defeats the purpose of the test
2. Try and keep your tests simple (but still thorough!), so you have to mock as little behaviour as possible
3. The **When/Then Cookbook** might help you: https://www.baeldung.com/mockito-behavior

Exercise 04

Code Coverage

1. No need to get 100% coverage
2. For every line/method, you should either cover it, or explain **why** you didn't cover it (e.g. "not covering trivial getters/setters")

Debugging

1. **Breakpoint.** Tell the debugger to halt here, as soon as it gets to this line. Add and remove breakpoints by left-clicking next to a line number.
2. **Current Position.** Program is currently halted on this line, the line hasn't yet been executed.
3. **Local Variables.** An overview of the current variable values.
4. **Call Stack.** The current method call stack.
5. **Navigation Tools.** Control where to go next (step over this line, step into it, etc.)
6. **Stop.** Stop the program, stop debugging.

Debugging

7. **Continue.** Continue running this program, either until it exists, or until it hits the next breakpoint.
8. **Debug Button.** Click this to run the program in debug mode. This will halt the program as soon as it hits a breakpoint. You can also debug a program by right-clicking on a main class, a test class or a test method, and clicking on "Debug As". We have already done this here, to get to this view.
9. **Java View vs. Debugger View**. Debug view (right button) is this view, Java view (left button) is the view you normally use when coding.

Debugging

Live DEMO:
Debugging the Snakes and Ladder Game

Checkers

Checkers

Definition

- The game of checkers/draughts is played on a standard chess board
- Black always goes first
- The 12 pieces of each player are always placed on the black spots nearest to him/her
- The pieces can only move diagonally
- The objective is to take all pieces from the opponent by jumping over these or make your opponent unable to move
- Further specifications can be found in the markdown files added to the exercise folder

Checkers Project

Tasks

- In the following weeks you will implement a completely functional checkers game
- This project is divided into three parts:
  - Create a skeleton of the Game with all classes and additionally a parser to parse a new gameboard and renderer to print the current gamestate
  - Implement movement and "King-Piece" logic
  - Implement the Final Game Logic (Winner/Looser)

Checkers (Notation)

#	:	Wall
O	:	Dark Square
_	:	Light Square
b	:	Black Piece
w	:	White Piece
B	:	Black King
W	:	White King

#	#	#	#	#	#	#	#	#	#
#	_	b	_	b	_	b	_	b	#
#	b	_	b	_	b	_	b	_	#
#	_	b	_	b	_	b	_	b	#
#	O	_	O	_	O	_	O	_	#
#	_	O	_	O	_	O	_	O	#
#	w	_	w	_	w	_	w	_	#
#	_	w	_	w	_	w	_	w	#
#	w	_	w	_	w	_	w	_	#
#	#	#	#	#	#	#	#	#	#

You could also implement your own representation if you think you have a better solution (but please state your changes, and provide your custom rendering scheme).

Your Task

Tasks

1. Set up the game representation (implement classes like **Game**, **Piece**, **Square** etc.)
2. Write a parser that reads the board specification. (There are already predefined boards in the 'games/' folder)
3. Write an ASCII renderer which prints any state of the gameboard (Use 'System.out.print' method)
4. Write tests so that the predefined boards are parsed correctly. (You can add more boards if you like)
5. Use the debugger and describe 3 problems that you solved using this tool
6. Create a class diagram of your implementation of the checkers game