So far you have survived 30 days of P2 so there is a high chance you are going to survive anything that’s to come.
The Key to success is Persistence.
Outline

Exercise 4 Recap

Debugging and Tools

Exercise 5 Outlook
Exercise 4 Recap

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
In `Game.java`:

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

Change to:

```java
public void play(IDie die) {
    ...
}
```
Then test with:

```java
@Before
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);
}
```
Another mocking example:

```java
@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`. 
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
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”)
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.
Debugger

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 the Turtle program
Exercise 5
Quoridor

- Simple board game
- At least 2 people
- Typically 9x9 tiles
- Each player starts on one edge of the wall, with the goal of getting to the opposite edge
- Each player gets a fixed number of walls (e.g. 5)
- In each turn, each player can either move (up, down, left, right) or place a wall
- Walls can only be placed such that after placing it, all players are still able to reach their destination
In our implementation: walls are placed on the tiles, rather than between them.
Player can’t move on a tile with a wall on it.
Boards are described in text files called *Board Files*:

```
width height numberOfwalls
player1 symbol1 x1 y1 target1
player2 symbol2 x2 y2 target2
... (additional players)
```

Example:

```
7 12 3
Otis Redding O 1 1 R
Solomon Burke S 7 12 L
```
Your Tasks

There will be three iterations. In this first iteration, you will have to:

- Implement and test a parser that parses a Game File and creates a Game instance from it
- Implement a renderer that prints the game state to the standard output
- Implement and test basic player movement
- **Special Tasks:** Use the debugger in at least three situations, take a screenshot each time, add it to a file called ‘debugger.md’, where you briefly explain how the debugger helped you there
When testing player movement, you shouldn’t rely on the renderer!

You don’t have to implement a fully functional game yet (no game logic, such as checking win conditions, valid moves, etc.)

Write proper JavaDoc class and method comments.

Do not forget about design by contract.