P2 - Exercise hour

Julius Oeftiger

2021-03-19

Outline

Ex. 3 Recap

Hooman

- Stores current position
- Executes commands

Parser

- Reads input and creates commands

Command classes

- CommandRight, -Up, ... implement ICommand
- Knowledge about the environment not needed!

```
class Environment {
  ...
  public Environment createFrom(String program) ... {
    ...
    // List<ICommand>
    var commands = CommandParser.parse(program);
    for (var cmd : commands) {
      hooman.executeCommand(cmd);
    }
    ...
  }
  ...
}
```

```
class Hooman {
    public void moveRight(int steps) {...}
    ...

    public void executeCommand(ICommand cmd) {
        cmd.execute(this);
    }
}
```

```
class Hooman {
    public void moveRight(int steps) {...}
    ...

    public void executeCommand(ICommand cmd) {
        cmd.execute(this);
    }
}
```

Note: The hooman executes any command according to the command supertype...

```
class CommandRight implements ICommand {
    private final int steps;

    public CommandRight(int steps) {
        this.steps = steps;
    }

    @Override
    public void execute(Hooman hooman) {
        hooman.moveRight(steps);
    }
}
```

... and commands select the correct "move" method.

```
class CommandRight implements ICommand {
    private final int steps;

        hooman.moveRight(steps);
    }
}
```

- Elegant way to avoid casting.
- Actual drawing takes place in hooman code.
- See "Design Patterns" book on course website (visitor pattern)

... and commands select the correct "move" method.

JUnit 5

```
@BeforeEach
public void initGame() {
    Player jack = new Player("Jack");
    Player jill = new Player("Jill");
    Game game = new Game(10, new Player[]{ jack, jill });

    assertTrue(game.notOver());
    assertTrue(game.firstSquare().isOccupied());
    assertEquals(1, jack.position());
    // ...
}
```

JUnit 5: Assertions

- import static org.junit.jupiter.api.Assertions.*;
  - Provides methods like *assertTrue*, *assertEquals*,...
- Import static allows you to use all the (static) assert methods without having to use a qualified name like:
  - Assertions.assertTrue(...) vs assertTrue(...)

JUnit 5: Assertions

```
assertTrue(condition);
assertEquals(expected, actual);

assertNull(object);
assertNotNull(object);

assertSame(expected, actual);
assertNotSame(expected, actual);
...
```

JUnit 5: Assertions

```
assertTrue(condition);
assertEquals(expected, actual);

assertNull(object);
assertNotNull(object);

assertSame(expected, actual);
assertNotSame(expected, actual);
...
```

~~*assert condition;*~~

Do not use the Java assertions!

JUnit 5: Assertions

```
assertTrue(jack.position() == 1);
```

JUnit 5: Assertions

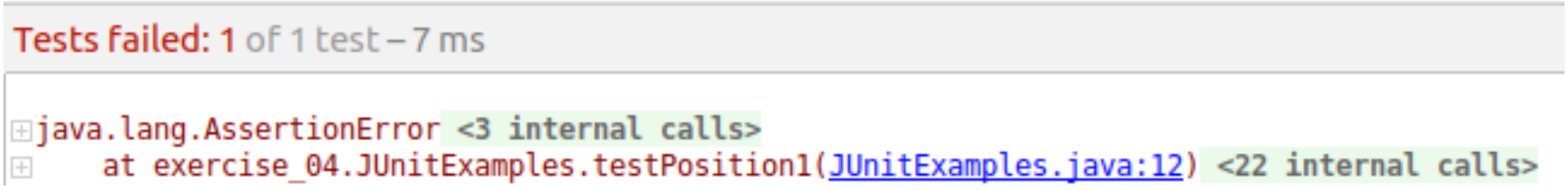
```
assertTrue(jack.position() == 1);
```

```
Tests failed: 1 of 1 test – 7 ms

java.lang.AssertionError <3 internal calls>
    at exercise_04.JUnitExamples.testPosition1(JUnitExamples.java:12) <22 internal calls>
```

JUnit 5: Assertions

```
assertTrue(jack.position() == 1);
```



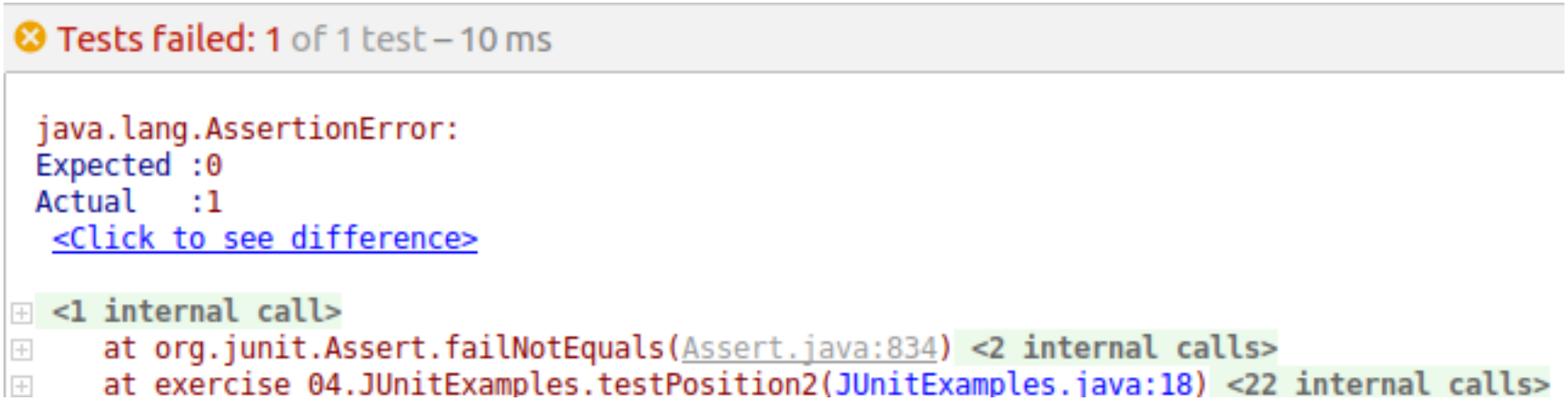
What went wrong? Need to check the code...

JUnit 5: Assertions

```
assertEquals(jack.position(), 1);
```

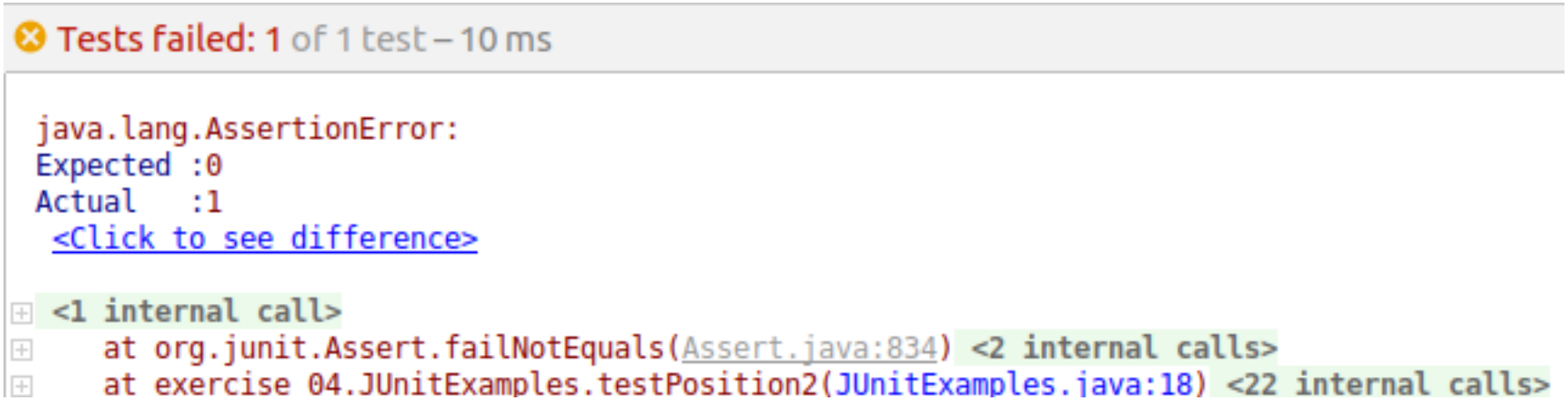
JUnit 5: Assertions

```
assertEquals(jack.position(), 1);
```



JUnit 5: Assertions

```
assertEquals(jack.position(), 1);
```



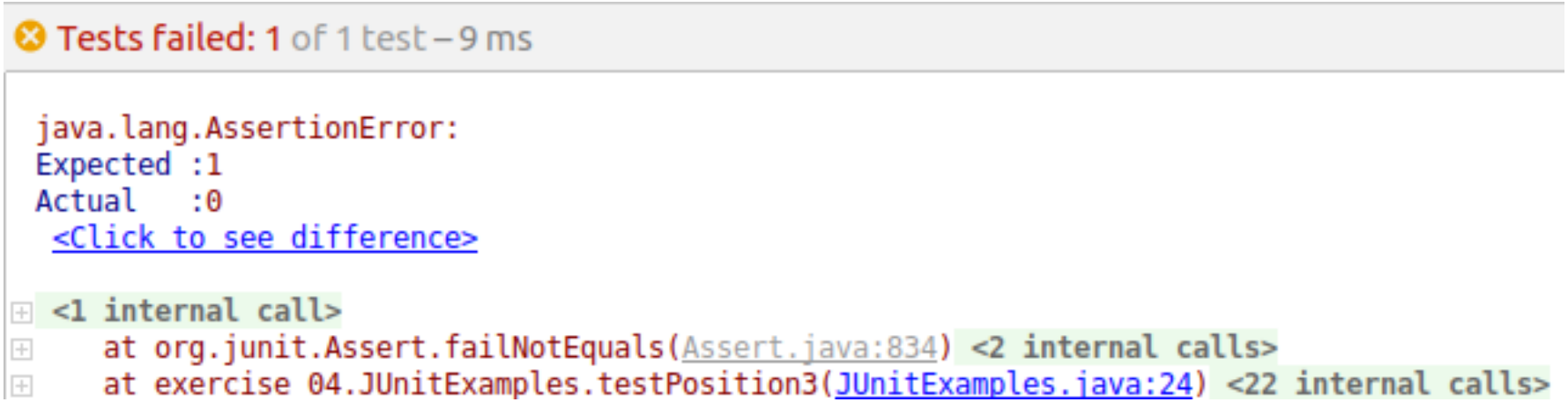
Wrong order: we expect 1, not 0!

JUnit 5: Assertions

```
assertEquals(1, jack.position());
```

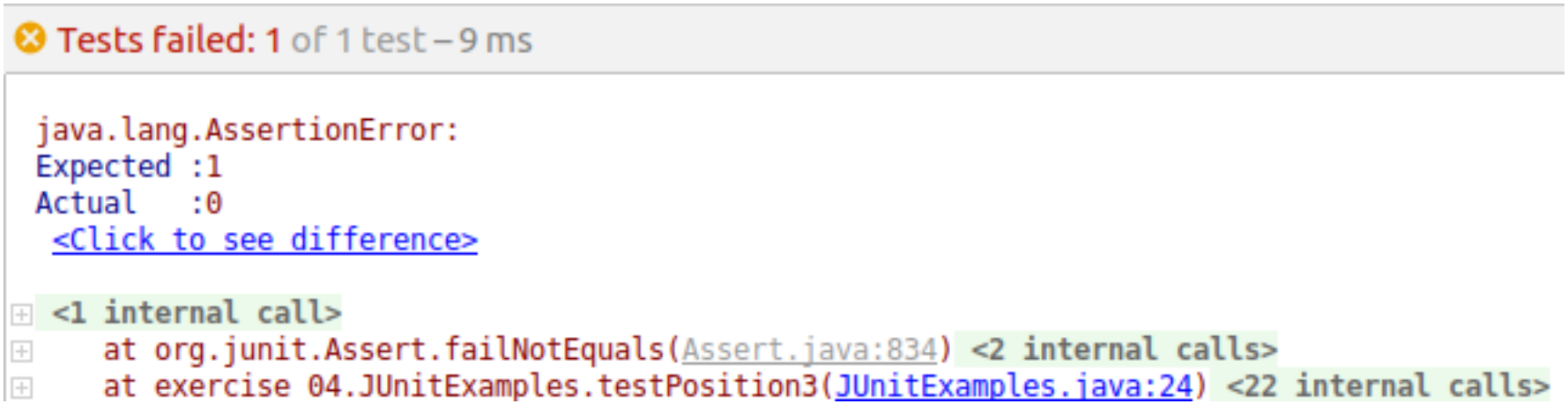
JUnit 5: Assertions

```
assertEquals(1, jack.position());
```



JUnit 5: Assertions

```
assertEquals(1, jack.position());
```



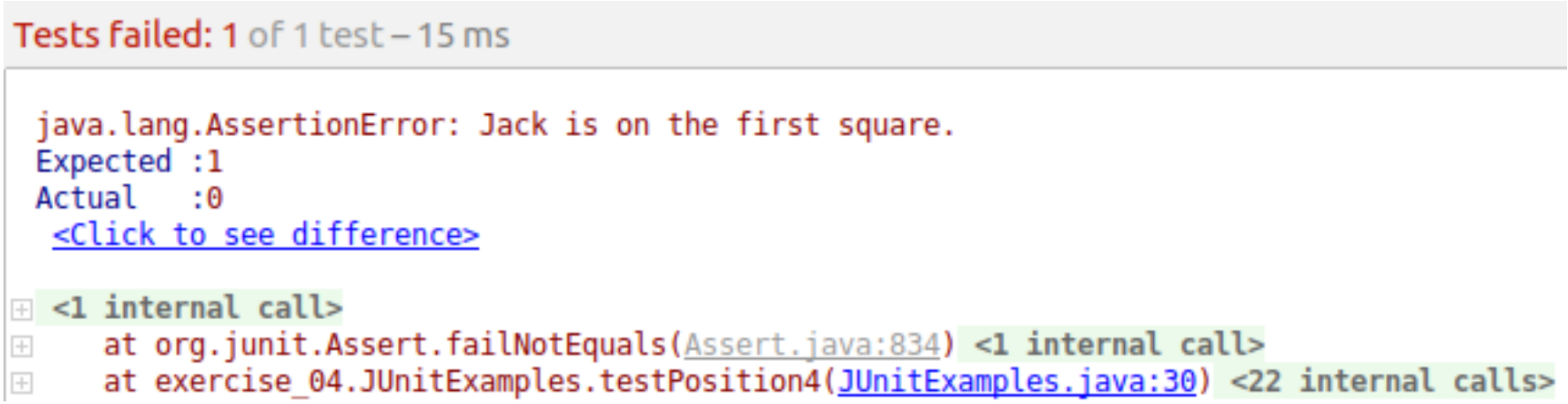
Correct order, but still unclear...

JUnit 5: Assertions

```
assertEquals(1, jack.position(),
             "Jack is on the first square.");
```

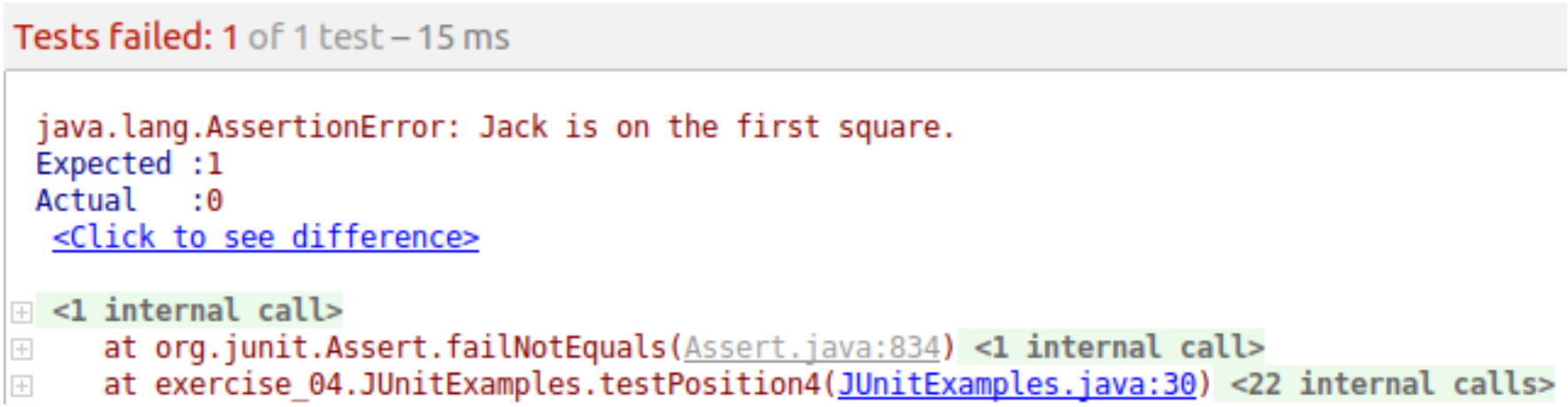
JUnit 5: Assertions

```
assertEquals(1, jack.position(),
             "Jack is on the first square.");
```



JUnit 5: Assertions

```
assertEquals(1, jack.position(),
             "Jack is on the first square.");
```



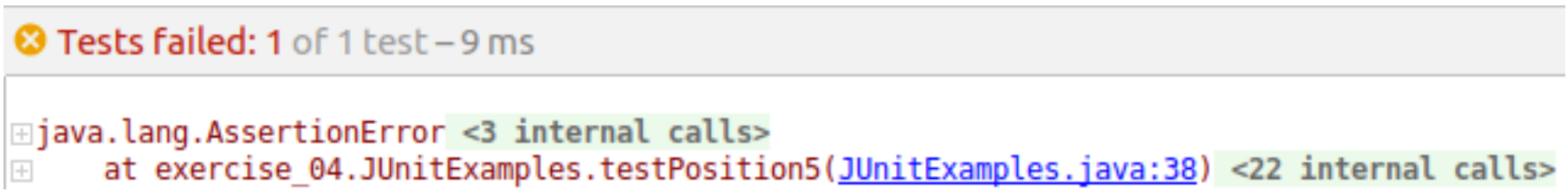
Provide a message (describing the expected outcome).

JUnit 5: Assertions

```
assertTrue(game.notOver() &&
           game.firstSquare().isOccupied() &&
           (1 == jack.position()) &&
           (1 == jill.position()));
```

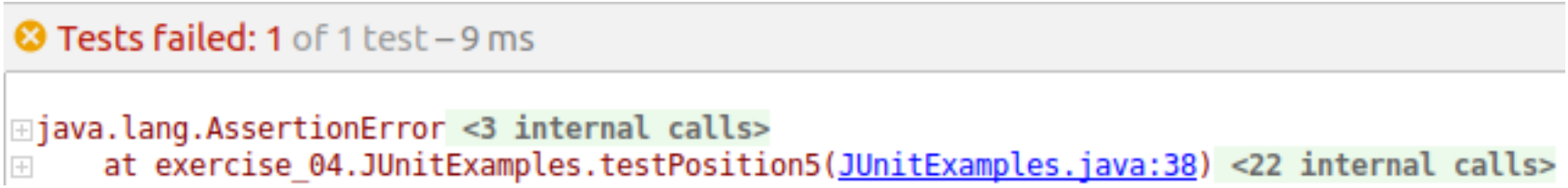
JUnit 5: Assertions

```
assertTrue(game.notOver() &&
           game.firstSquare().isOccupied() &&
           (1 == jack.position()) &&
           (1 == jill.position()));
```



JUnit 5: Assertions

```
assertTrue(game.notOver() &&
           game.firstSquare().isOccupied() &&
           (1 == jack.position()) &&
           (1 == jill.position()));
```



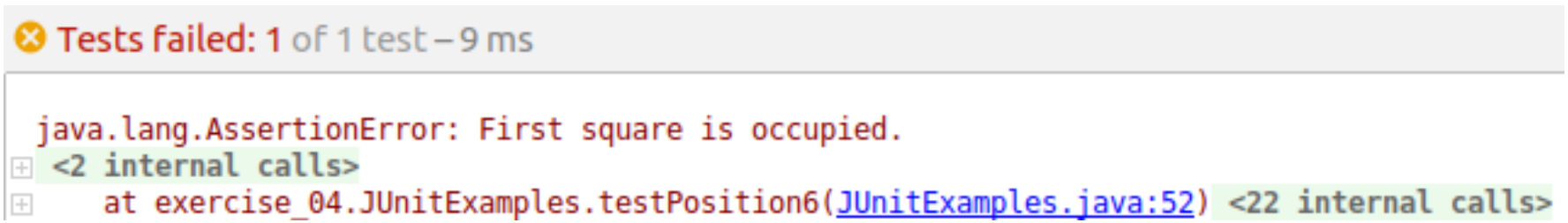
Which condition made the assertion fail?

JUnit 5: Assertions

```
assertTrue(game.notOver(),
           "Game is not over.");
assertTrue(game.firstSquare().isOccupied(),
           "First square is occupied.");
assertEquals(1, jack.position(),
             "Jack is on the first square.");
assertEquals(1, jill.position(),
             "Jill is on the first square.");
```

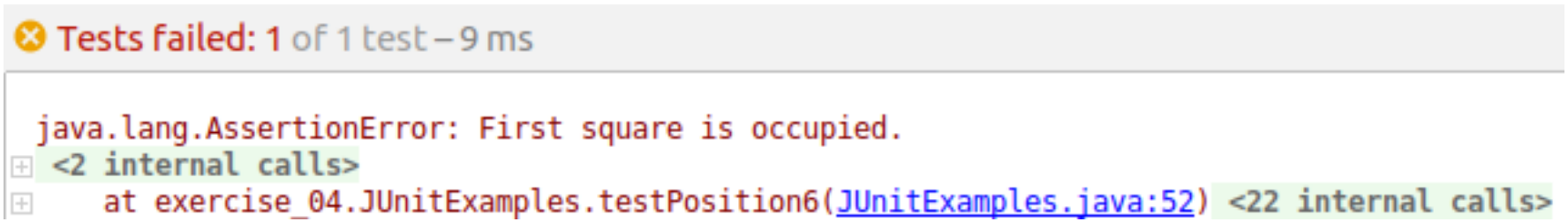
JUnit 5: Assertions

```
assertTrue(game.notOver(),
           "Game is not over.");
assertTrue(game.firstSquare().isOccupied(),
           "First square is occupied.");
assertEquals(1, jack.position(),
             "Jack is on the first square.");
assertEquals(1, jill.position(),
             "Jill is on the first square.");
```



JUnit 5: Assertions

```
assertTrue(game.notOver(),
           "Game is not over.");
assertTrue(game.firstSquare().isOccupied(),
           "First square is occupied.");
assertEquals(1, jack.position(),
             "Jack is on the first square.");
assertEquals(1, jill.position(),
             "Jill is on the first square.");
```



Use one condition per assertion!

JUnit 5: Initialization

```
private Game game;
private Player jack, jill;

@BeforeEach
public void initNewGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    game = new Game(10, new Player[] { jack, jill });
}
```

JUnit 5: Initialization

```
private Game game;
private Player jack, jill;

@BeforeEach
public void initNewGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    game = new Game(10, new Player[] { jack, jill });
}
```

Use *@BeforeEach* to initialize a new game
before each test method.

JUnit 5: Setup & Teardown

- *@BeforeEach*
  - Initialize before each test
  - Use for creating things common to all tests
- *@AfterEach*
  - Clean up after each test
  - Executed even if *@BeforeEach* or *@Test* fails

JUnit 5: Setup & Teardown

- *@BeforeEach*
  - Initialize before each test
  - Use for creating things common to all tests
- *@AfterEach*
  - Clean up after each test
  - Executed even if *@BeforeEach* or *@Test* fails
- *@BeforeAll* (must be *static*)
  - Called once before any *@Test* method is executed.
  - Use for time intensive tasks (creating database connections, ...)
- *@AfterAll* (must be *static*)
  - Clean up after all tests were run

JUnit 5: Testing Exceptions

- Make sure an exception is thrown
- Useful for making sure errors (e.g. bad input) are actually detected and handled correctly

```
@Test
public void negativeMoveIsIllegal()  {
    assertThrows(IllegalMoveException.class,
                 () -> hooman.moveRight(-1));
}
```

JUnit 5: Testing Performance

- Testing execution speed using the *timeout* parameter
- Time in milliseconds (!)

```
@Test
@TimeOut(1)
// @TimeOut(value=1, TimeUnit.SECONDS)     [equivalent]
public void hoomanIsFast() {
    hooman.moveRight(10);
}
```

JUnit 5

- **No** control over order of execution (!)

JUnit 5

- **No** control over order of execution (!)
- Tests should not depend on other tests

JUnit 5

- **No** control over order of execution (!)
- Tests should not depend on other tests
- Do not share data between tests (instance variables, ...)

Writing good tests

- Consider different inputs and parameters
  - Common inputs, values raising exceptions
- Boundary values
  - "off-by-one" errors
- Uncommon values
  - *null* (if allowed by the contracts)
  - Empty list, array, ...
- Test outputs
  - Returned values and exceptions
- Test side effects
  - State of the system

Writing good tests

- Test classes should thoroughly test a single class
- Write test **during** development
  - You can write them even before you implemented the functionality. You know you're done, when all tests pass.
- Write tests for every feature

Writing good tests

- As with all code: **make it readable**
  - Proper, self-explaining naming
  - JavaDoc if needed
  - Use the appropriate assertions
  - Keep tests short (few assertions per method)

Mocking

- Some components may be hard to test
  - Non-deterministic results (e.g. a die)
  - Behaviour that is difficult to reproduce (e.g. network failures)
  - Slow or expensive components
  - Incomplete components

Mocking

- Some components may be hard to test
  - Non-deterministic results (e.g. a die)
  - Behaviour that is difficult to reproduce (e.g. network

Let's just fake it!

Mocking

- Mock objects: Crash test dummies for programmers
- Fake the real thing by manually specifying the behaviour
- Use in place of real objects

Mocking

```
// you can mock concrete classes, not only interfaces
ArrayList mockedList = mock(ArrayList.class);
```

Create a mock object
$\rightarrow$ it can be used like any other object of that type

[0]Code from https://site.mockito.org/#how

Mocking

```
// you can mock concrete classes, not only interfaces
ArrayList mockedList = mock(ArrayList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");
```

Tell the mock object how to behave.
Here: when *get(0)* is called, return the String *"first"*.

[0]Code from https://site.mockito.org/#how

Mocking

```
// you can mock concrete classes, not only interfaces
ArrayList mockedList = mock(ArrayList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

// the following prints "first"
System.out.println(mockedList.get(0));

// the following prints "null",
// because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

[0]Code from https://site.mockito.org/#how

Mocking

```
// you can mock concrete classes, not only interfaces
ArrayList mockedList = mock(ArrayList.class);
```

Read the documentation!
http://javadoc.io/page/org.mockito/mockito-core/latest/org/mockito/Mockito.html

```
System.out.println(mockedList.get(0));

// the following prints "null",
// because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

[0]Code from https://site.mockito.org/#how

Mockito: Example

```
public ISquare moveAndLand(int moves) {
    assert moves >= 0;
    return game.findSquare(position, moves)
               .landHereOrGoHome());
}
```

```
@Test
public void testMoveAndLand() {
    Game game = new Game(10, new Player("Jack"));
    ISquare start = game.getSquare(2);
    ISquare destination = startSquare.moveAndLand(2);
    assertEquals(game.getSquare(4), destination);
}
```

Mockito: Example

```
public ISquare moveAndLand(int moves) {
    assert moves >= 0;
    return game.findSquare(position, moves)
               .landHereOrGoHome());
}
```

```
@Test
public void testMoveAndLand() {
    ...
}
```

Also needs *Game.getSquare*, *Game.findSquare* and *ISquare.LandHereOrGoHome* to work properly!

Mockito: Example

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);

    when(game.isValidPosition(anyInt())).thenReturn(true);


    ISquare testSquare = new Square(game, 1);
    ISquare finding = mock(Square.class);
    ISquare landing = mock(Square.class);


    when(game.findSquare(1, 2)).thenReturn(finding);
    when(finding.landHereOrGoHome()).thenReturn(landing);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(landing, destination);
}
```

Mockito: Example

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class); // create fake Game
    // tell the game mock what to do if isValidPosition() is called
    when(game.isValidPosition(anyInt())).thenReturn(true);


    ISquare testSquare = new Square(game, 1);
    ISquare finding = mock(Square.class);
    ISquare landing = mock(Square.class);


    when(game.findSquare(1, 2)).thenReturn(finding);
    when(finding.landHereOrGoHome()).thenReturn(landing);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(landing, destination);
}
```

Mockito: Example

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class); // create fake Game
    // tell the game mock what to do if isValidPosition() is called
    when(game.isValidPosition(anyInt())).thenReturn(true);

    // create target on which we want to test moveAndLand()
    ISquare testSquare = new Square(game, 1);
    ISquare finding = mock(Square.class); // mocks for findSquare()
    ISquare landing = mock(Square.class); // and landHereOrGoHome()


    when(game.findSquare(1, 2)).thenReturn(finding);
    when(finding.landHereOrGoHome()).thenReturn(landing);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(landing, destination);
}
```

Mockito: Example

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class); // create fake Game
    // tell the game mock what to do if isValidPosition() is called
    when(game.isValidPosition(anyInt())).thenReturn(true);

    // create target on which we want to test moveAndLand()
    ISquare testSquare = new Square(game, 1);
    ISquare finding = mock(Square.class); // mocks for findSquare()
    ISquare landing = mock(Square.class); // and landHereOrGoHome()

    // mock behaviour of game and finding
    when(game.findSquare(1, 2)).thenReturn(finding);
    when(finding.landHereOrGoHome()).thenReturn(landing);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(landing, destination); // actual test for testSquare
}
```

Exercise 4

- Test Snakes & Ladders (our implementation)
- Fix exercise 3 if necessary
- Use JUnit and Mockito
- Write good tests with code coverage and qualitative criteria in mind
- See *exercise_04.md* for more details