P2: Exercise 3 Discussion

Claudio Corrodi

Exercise 3

- Parser
  - Read input, create Command objects
- Command classes
  - **CommandLeft**, **CommandUp**, **…**, extend **Command**
  - Don’t need to know about the board at all
- Turtle
  - Stores current position
  - Executes command

Exercise 3: Double Dispatch

```
class Turtle {
    public void moveRight(CommandRight command) {}
    public void moveUp(CommandUp command) {}
    public void jump(CommandJump command) {}
    public void executeCommand(Command command) {
        command.execute(this);
    }
}

class CommandRight {
    public void execute(Turtle turtle) {
        turtle.moveRight(this);
    }
}
```

Exercise 3: Double Dispatch

```
class Turtle {
    public void moveRight(CommandRight command) {}
    public void moveUp(CommandUp command) {}
    public void jump(CommandJump command) {}
    public void executeCommand(Command command) {
        command.execute(this);
    }
}

class CommandRight {
    public void execute(Turtle turtle) {
        turtle.moveRight(this);
    }
}
```

1. Execute any command (use supertype)

Exercise 3: Double Dispatch

```
class Turtle {
    public void moveRight(CommandRight command) {}
    public void moveUp(CommandUp command) {}
    public void jump(CommandJump command) {}
    public void executeCommand(Command command) {
        command.execute(this);
    }
}

class CommandRight {
    public void execute(Turtle turtle) {
        turtle.moveRight(this);
    }
}
```

2. Commands select the correct “move…” method.

Exercise 3: Double Dispatch

```
class Turtle {
    public void moveRight(CommandRight command) {}
    public void moveUp(CommandUp command) {}
    public void jump(CommandJump command) {}
    public void executeCommand(Command command) {
        command.execute(this);
    }
}

class CommandRight {
    public void execute(Turtle turtle) {
        turtle.moveRight(this);
    }
}
```

3. The turtle knows that a CommandRight needs to be executed.

Exercise 3: Double Dispatch

```
class Turtle {
    public void moveRight(CommandRight command) {}
    public void moveUp(CommandUp command) {}
    public void jump(CommandJump command) {}

List<Command> commands = parser.parse(program);
for (Command c : commands) {
    turtle.executeCommand(c);
}

    public void execute(Turtle turtle) {
        turtle.moveRight(this);
    }
}
```

Exercise 3: Double Dispatch

```
class Turtle {
        turtle.moveRight(this);
    }
}
```

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

P2: Unit Testing

Claudio Corrodi

JUnit 4

```
@Test
public void gameInitialization() {
    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 4

```
@Test
public void gameInitialization() {
    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());
    // ...
}
```

Annotate test methods with @Test

JUnit 4

```
@Test
public void gameInitialization() {
    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());
    // ...
}
```

Use assertions to test the state of the program

JUnit 4: Assertions

- `import static org.junit.Assert.*;`
  - Provides methods like “assertTrue”, “assertEquals”, …
- NB: Import static allows you to use the (static) Assert methods without having to use a qualified name
  - `Assert.assertTrue(…)` vs `assertTrue(…)`

JUnit 4: Assertions

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

assertNull(object);
assertNotNull(object);

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

assertArrayEquals(boolean[] expected, boolean[] actual)
```

→ See class org.junit.Assert for more!

JUnit 4: Assertions

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

assertNull(object);
assertNotNull(object);

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

assertArrayEquals(boolean[] expected, boolean[] actual)
```

→ See class org.junit.Assert for more!

~~`assert condition;`~~

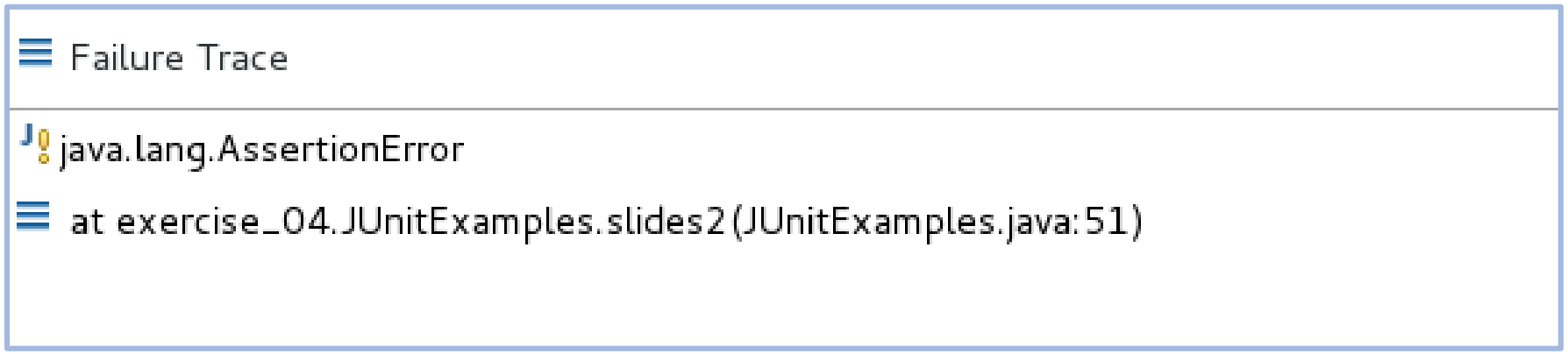
Do not use the Java assertions (using the assert *keyword*)!

JUnit 4: Assertions

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

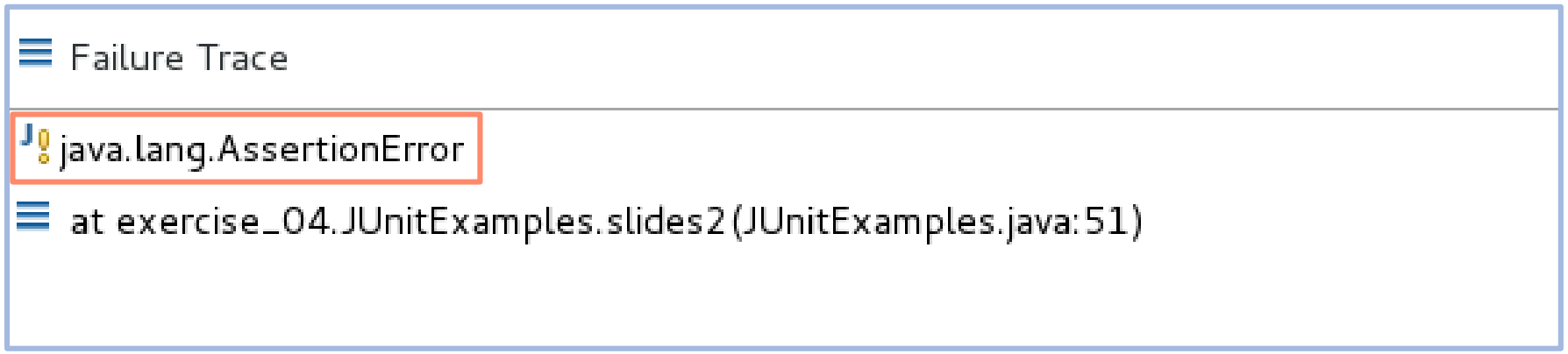
JUnit 4: Assertions

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



JUnit 4: Assertions

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



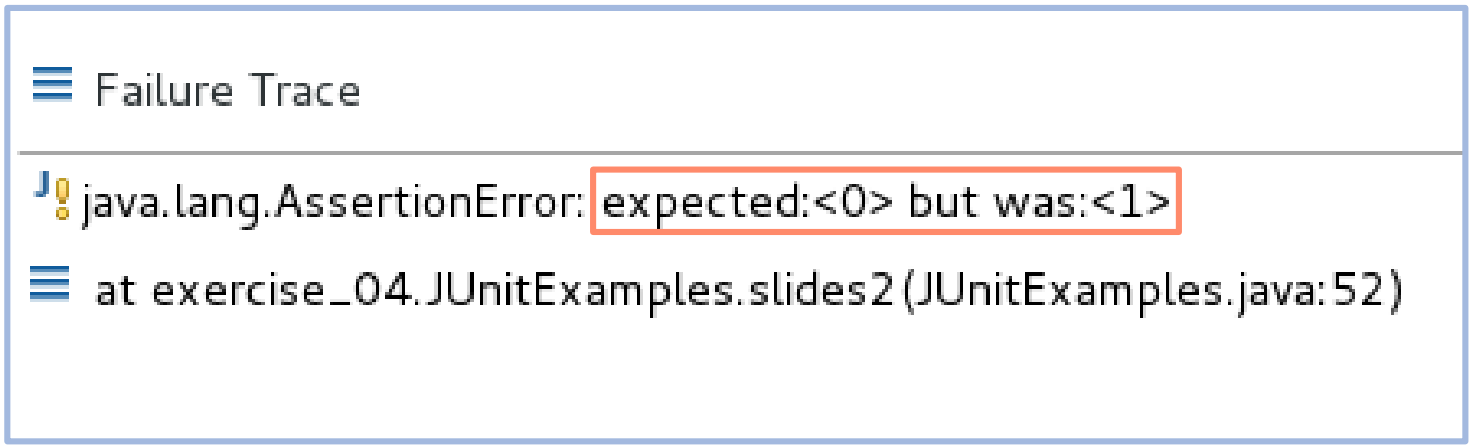
What went wrong? Need to check the code…

JUnit 4: Assertions

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

JUnit 4: Assertions

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



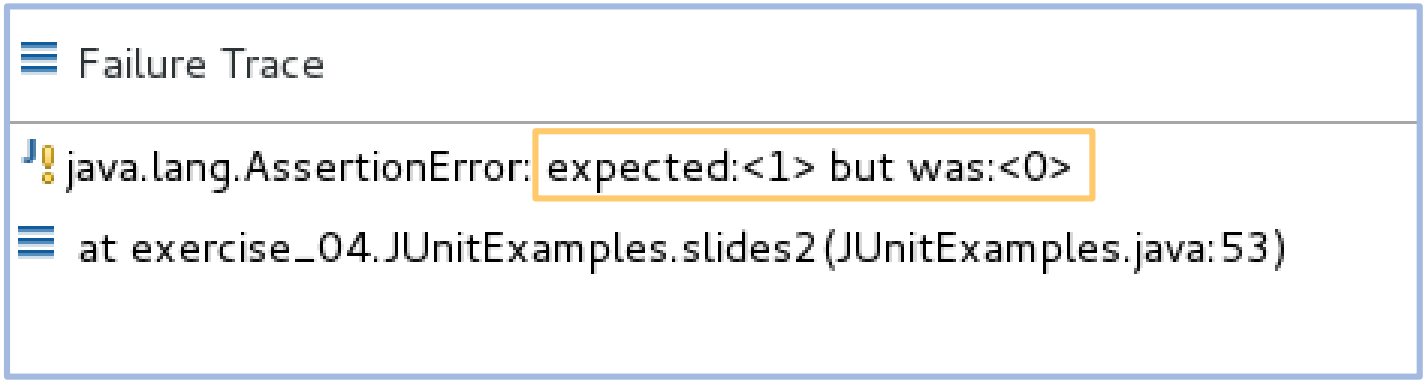
Wrong order: we expect 1, not 0!

JUnit 4: Assertions

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

JUnit 4: Assertions

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



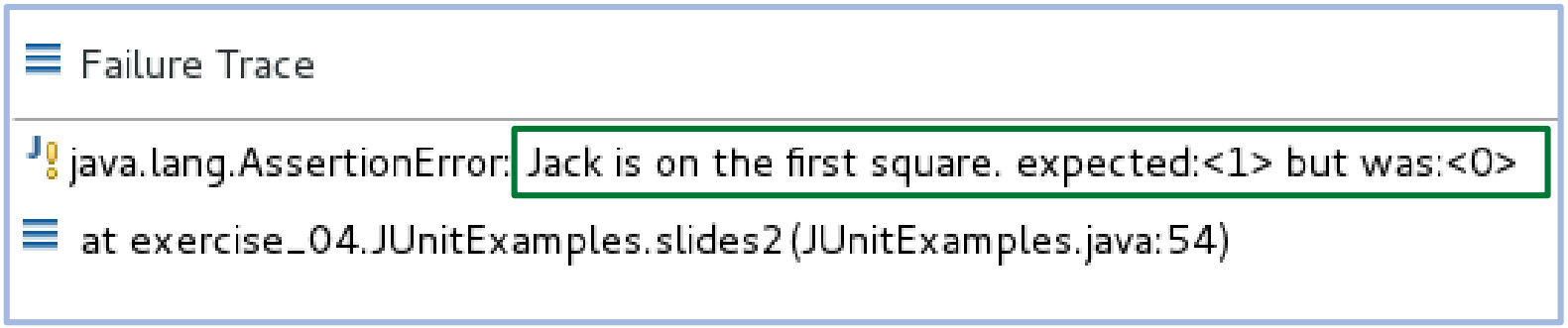
Correct order, but still unclear…

JUnit 4: Assertions

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

JUnit 4: Assertions

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



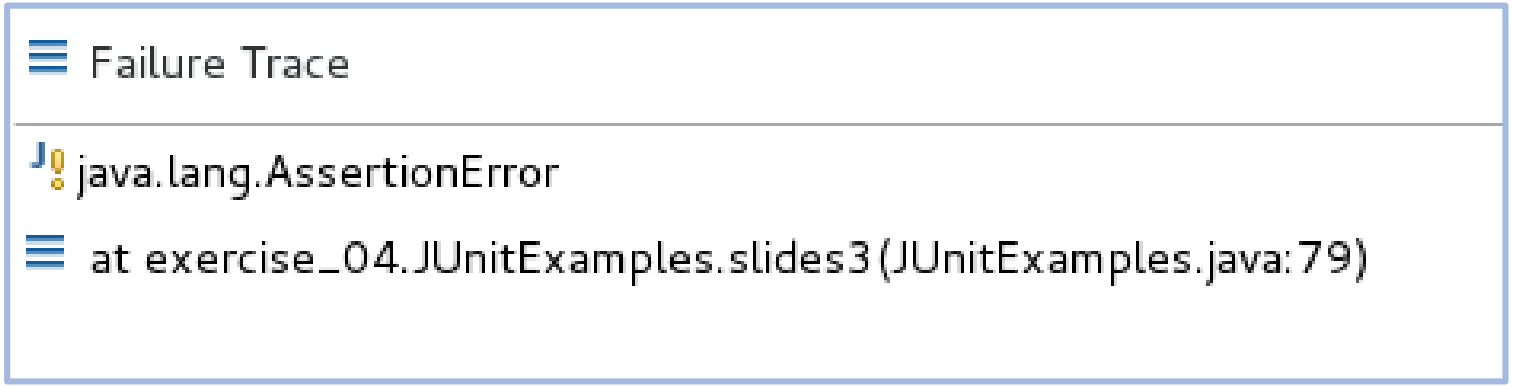
Provide a message (describing the expected outcome) as first argument.

JUnit 4: Assertions

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

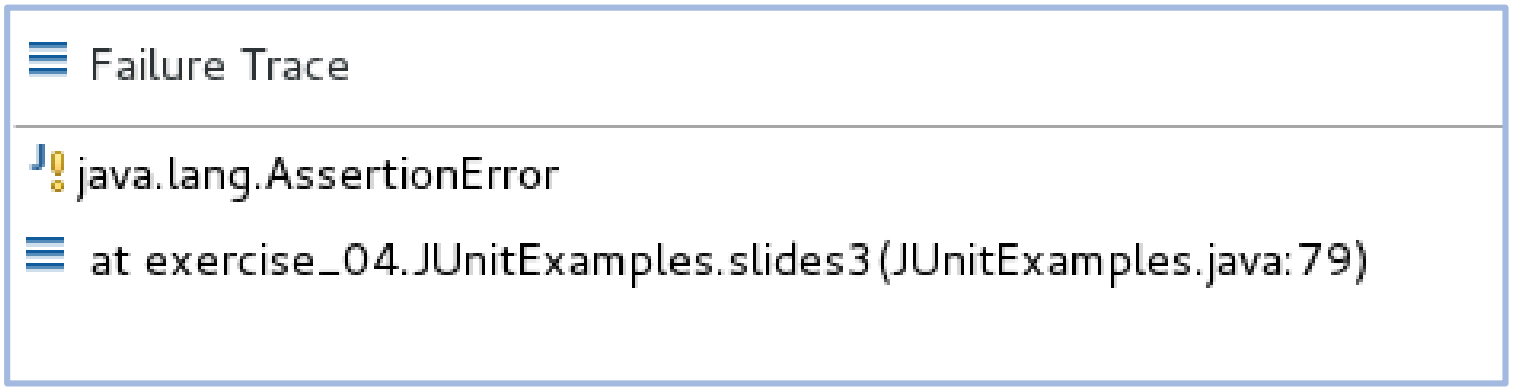
JUnit 4: Assertions

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



JUnit 4: Assertions

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



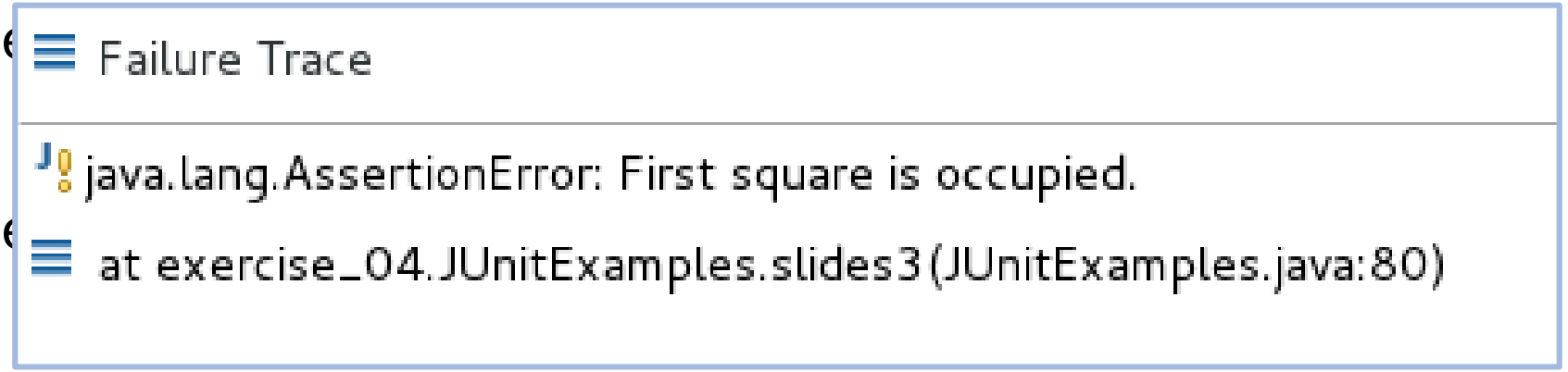
Which condition made the assertion fail?

JUnit 4: Assertions

```
assertTrue("Game is not over.",
           game.notOver());

assertTrue("First square is occupied.",
           game.firstSquare().isOccupied());

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

assertEquals("Jill is on the first square.",
             1, jill.position());
```

JUnit 4: Assertions

```
assertTrue("Game is not over.",
           game.notOver());

assertTrue("First square is occupied.",
           game.firstSquare().isOccupied());

asse
```




```
asse
```

Use one condition per assertion!

JUnit 4: Initialization

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

    assertEquals(1, jill.position());
}
```

JUnit 4: Initialization

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

    assertEquals(1, jill.position());
}
```

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

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

JUnit 4: Initialization

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

    assertEquals(1, jill.position());
}
```

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

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

Duplicate code for initializing a new game!

JUnit 4: Initialization

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

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

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

JUnit 4: Initialization

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

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

```
@Test
public void initialPositionJill() {
    assertEquals(1, jill.position());
}
```

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

JUnit 4: Initialization

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

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

```
@Test
public void initialPositionJill() {
    assertEquals(1, jill.position());
}
```

```
@Test
public void initialPositionJack() {
    assertEquals(1, jack.position());
}
```

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

JUnit 4: Setup & Teardown

- @Before
  - Executed before each test method
  - Use for initializing things common to all tests
  - E.g. Snakes & Ladders game, opening a configuration file

- @After
  - Clean up after tests
  - Executed even if @Before or @Test fails
  - E.g. closing a file, clearing a cache

JUnit 4: Setup & Teardown

- @BeforeClass
  - Executed once per class, before any @Test method is executed
  - Use for time intensive tasks, e.g. connecting to a database
- @AfterClass
  - Executed once per class, after all @Test methods have been executed
  - Useful for cleaning up resources, e.g. closing the database connection
- Both must be static methods

JUnit 4: Test Suites

Group tests using test suites

```
import org.junit.runners.Suite;
import org.junit.runner.RunWith;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        SquareInitialization.class,
        PlayerInitialization.class,
        BasicGameStateInitialization.class})
public class Initialization {
    //nothing
}
```

- Use test classes to verify **units** (methods, classes)
- Use test suites to verify **features**

JUnit 4: Testing Exceptions

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

```
@Test(expected=IllegalMoveException.class)
public void negativeMoveIsIllegal() throws IllegalMoveException {
    turtle.moveRight(-1);
}
```

JUnit 4: Testing Performance

- Testing execution speed using the “timeout” parameter

- Time in milliseconds

```
@Test(timeout=10)
public void turtleIsFast() {
    turtle.moveLeft(10);
}
```

JUnit 4

- No control over order of execution
- Tests should not depend on other tests
- Do not share data between tests (instance variables, files, databases, …)

JUnit 4

```
@BeforeClass
public static void newGame() {
    game = new Game(...);
   // initialize a new game
}

@Test
public void moveJack() {
   game.movePlayer(2);
   // assertions
}

@Test
void moveJill() {
   game.movePlayer(4);
   // assertions
}
```

JUnit 4

```
@BeforeClass
public static void newGame() {
    game = new Game(...);
   // initialize a new game
}

@Test
public void moveJack() {
   game.movePlayer(2);
   // assertions
}

@Test
void moveJill() {
   game.movePlayer(4);
   // assertions
}
```

Create a new game before any tests are executed, then execute moveJack, followed by moveJill

JUnit 4

```
@BeforeClass
public static void newGame() {
    game = new Game(...);
    // initialize a new game
}

@Test
public void moveJack() {
    game.movePlayer(2);
    // assertions
}

@Test
void moveJill() {
    game.movePlayer(4);
    // assertions
}
```

Create a new game before any tests are executed, then execute moveJack, followed by moveJill

WRONG!

moveJill might be executed before moveJack!

JUnit 4

```
@Before
public void newGame() {
    game = new Game(...);
   // initialize a new game
}

@Test
public void moveJack() {
   game.movePlayer(2);
   // assertions
}

@Test
void moveJill() {
   game.movePlayer(2);
   game.movePlayer(4);
   // assertions
}
```

- Now the tests are independent
- But we just copied the code from moveJack
- If moveJack fails, so does moveJill

~~JUnit 4~~ JExample

```
@Test
public void newGame() {
    game = new Game(...);
   // initialize a new game
}

@Given("#newGame")
public void moveJack() {
   game.movePlayer(2);
   // assertions
}

@Given("#moveJack")
void moveJill() {
   game.movePlayer(4);
   // assertions
}
```

JExample: Short test methods, ordered execution.

JExample

- JExample != JUnit
- JExample allows you to specify dependencies between test methods
- Allows you to avoid tests with a large number of assertions
- Execution stops when a test fails
  - No need to find out which test made every other test fail

Writing good tests

- Consider different inputs and parameters
  - Common inputs
  - Boundary values, corner cases
  - Values raising exceptions
- Test outputs
  - Returned values and exceptions
- Test side effects
  - State of the system

Writing good tests

- Boundary values
  - Find “off-by-one” errors
  - Turtle game: -1, 0, 1, 100, 101, …
- Uncommon values
  - null (if allowed by the contracts)
  - empty list, array, …
- Invalid inputs
  - **But not** values violating the preconditions

Writing good tests

- Test classes should thoroughly test a single class
- Write tests during development
  - You can write them even before you implemented the functionality. Then 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 (not just “assertTrue” for everything)
  - 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. networks failures)
  - Slow or expensive components (e.g. setting up a database)
  - Incomplete components (e.g. class that’s specified but not implemented yet)

Mocking

- Some components may be hard to test
  - Non-deterministic results (e.g. a die)
  - Beh… …etworks failu…
  - Slow or expensive components (e.g. setting up a database)
  - Incomplete components (e.g. class that's specified but not implemented yet)

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

Mockito: A mocking framework

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

Create a mock object

→ it can be used like any other object of that type

Code from http://site.mockito.org/#how

Mockito: A mocking framework

```
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.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”.

Code from http://site.mockito.org/#how

Mockito: A mocking framework

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

Use the object like any other!

Code from http://site.mockito.org/#how

Mockito: A mocking framework

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

// stubbing appears before the actual execution
w

/
S

/                                                    bbed
System.out.println(mockedList.get(999));
```

Go read the documentation…

http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html

Code from http://site.mockito.org/#how

Mockito: Example

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

Mockito: Example

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

```
@Test
public void testMoveAndLand() {
    Player jack = new Player("Jack");
    Player jill = new Player("Jill");
    Player[] args = {jack, jill};
    Game game = new Game(12, args);
    ISquare startSquare = 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();
}
```

Also needs Game.getSquare, Game.findSquare, and Square.landHereOrGoHome to work properly!

```
@Test
public void testMoveAndLand() {
    Player jack = new Player("Jack");
    Player jill = new Player("Jill");
    Player[] args = {jack, jill};
    Game game = new Game(12, args);
    ISquare startSquare = 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 testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(true);
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

Mockito: Example

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

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(true);
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

create a fake Game

Mockito: Example

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

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(true);
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

tell the game mock what to do when “isValidPosition” is called

Mockito: Example

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

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(true);
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

testSquare is the target on which we want to test “moveAndLand”

Mockito: Example

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

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(true);
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

start and stop are the square mocks we use for “findSquare” and “landHereOrGoHome”

Mockito: Example

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

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(t
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

mock behaviour of game and start

Mockito: Example

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

```
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class);
    ISquare testSquare;
    ISquare start, stop;
    when(game.isValidPosition(anyInt())).thenReturn(t
    testSquare = new Square(game, 1);
    start = mock(Square.class);
    stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

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

actual test
calls “moveAndLand” on
testSquare, but uses mocks
for everything else

Exercise 4

- Test previous games
  - Snakes & Ladders and Turtle Game
- Use JUnit, JExample, and Mockito
- Write good tests with **code coverage** and qualitative criteria in mind
- See exercise_04.md for more details!