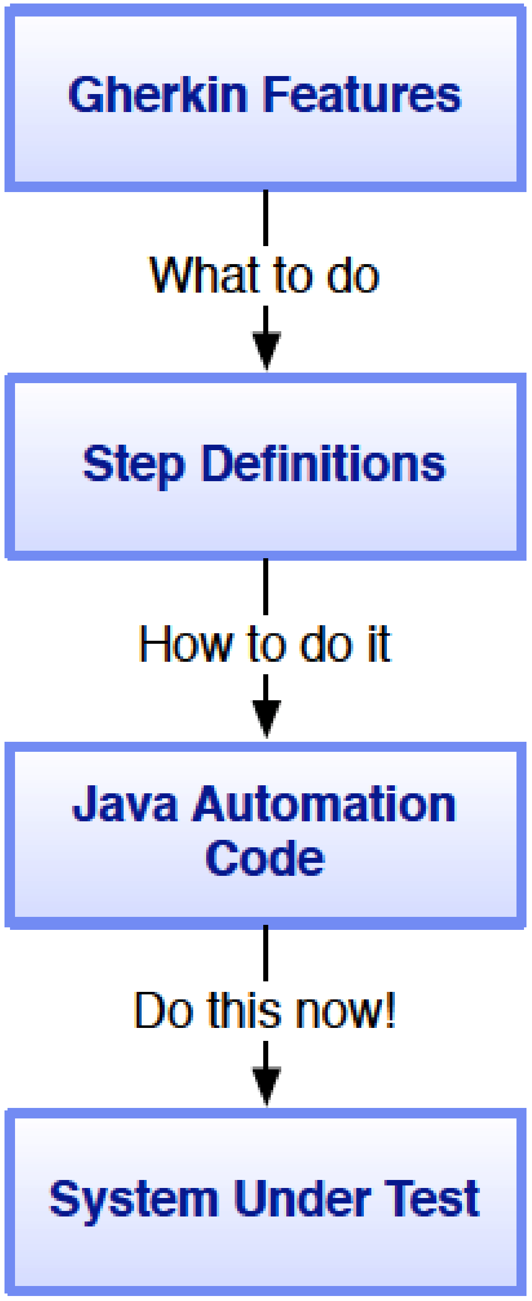
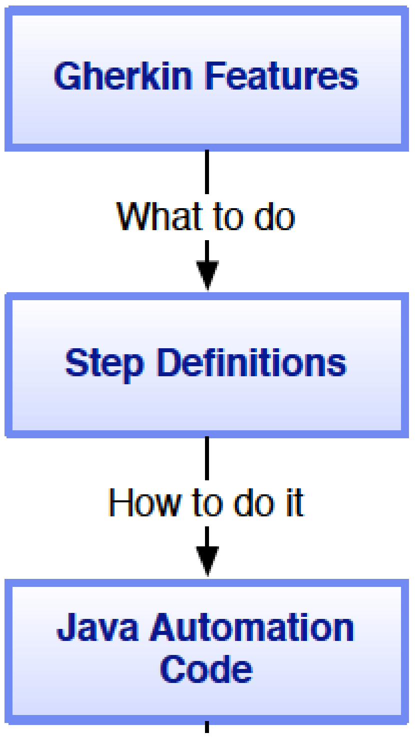
From what to how

- Given I have $100 in my Account
  - -> how to make the following things happen?
  - Create an account for the protagonist in the scenario
  - Set the balance of that account to be $100
- How is this achieved?


Gherkin Features
What to do
Step Definitions
How to do it
Java Automation Code
Do this now!
System Under Test

Two sides of a stepdefinition



- On the outside -> translates plain language into code
- inside -> tells system what to do using automation code

Step versus Stepdefinition

- A scenario is made up of a series of steps
  - written in plain language
  - is just documentation
- A step definition is a piece of code -> says to Cucumber
  - “If you see a step that looks like this"
  - "Then execute this piece of code"

How steps are matched

1. steps are expressed in plain text
2. Cucumber scans the text of each step for patterns
3. Patterns are described using regexp's

An example

```
Feature: Cash withdrawal
    Scenario: Successful withdrawal from an account in credit
        Given I have $100 in my account
        When I request $20
        Then $20 should be dispensed
```

- Cucumber handles: Given I have $100 in my account
- it asks: is there any stepdefinition matching this step?
- i.e. regexp: I have \\$100 in my account -> matches this step
- if it finds a stepdefinition with this regexp it will execute it

Creating a Step Definition

- Step definitions live in ordinary files
- In Java Annotations like @Given are used

```java
@Given("I have \\$100 in my Account")
public void iHave$100InMyAccount() throws Throwable {
  // TODO: code that puts $100 into User's Account goes here
}
```

- Suggestion keep a separate file per domain entity
  - keep step definitions that work with similar parts of the system together

From cmdline to IDE

```
<properties>
    <cucumber.version>1.2.5</cucumber.version>
    <junit.version>4.12</junit.version>
</properties>
<dependencies>
    <dependency>
        <groupId>info.cukes</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>${cucumber.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>info.cukes</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>${cucumber.version}</version>
        <scope>test</scope>
```

- See appendix

From cmdline to IDE

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<version>2.12.2</version>
<configuration>
<argLine>-Duser.language=en</argLine>
<argLine>-Xmx1024m</argLine>
<argLine>-XX:MaxPermSize=256m</argLine>
<argLine>-Dfile.encoding=UTF-8</argLine>
<useFile>false</useFile>
</configuration>
</plugin>
```

- See appendix

Given, When, Then Are the Same

- Cucumber:
- ignores the keyword when matching a step
  - all of the annotations are aliases for StepDefAnnotation
- annotations are just there for extra documentation to express the intent of each step definition
  - a step definition will match any Gherkin step as long as the regular expression matches the main text of the step.

Difficult to spot

- After some weeks you add a new scenario to the feature:

```
Scenario: New accounts get a $1 gift
    Given I have a brand new Account
    And I deposit $99
    Then I have $100 in my Account
```

- Compare Then step with Given from former scenario:

```
Given I have $100 in my account
```

- **Both are mapped by the same regexp!**

Be careful with regexp

- Cucumber ignores the @Given/@When/@Then annotation when matching a step
- We saw -> a false positive: passing when it should have been failing
- To avoid this -> pay careful attention to the precise wording in steps

```
Given I have deposited $100 in my Account
Then the balance of my Account should be $100
```

Capturing Arguments

- To capture arguments use the flexibility of regular expressions
  - capture groups -> ( )
  - wildcards -> ? * . among others

Capture Groups -> ()

- Change \\$100– >$(100) in

```
@Given("I have deposited \\$100 in my Account")
public void iHaveDeposited$100InMyAccount() {
    // TODO: code goes here
}

@Given("I have deposited \\$(100) in my Account")
public void iHaveDeposited$100InMyAccount(int amount) {
    // TODO: code goes here
}
```

- This still works only for $100 -> make it more flexible

Creating more flexibility

Joe asks:

What If I Actually Want to Match a Dot?

Any of the metacharacters like the dot can be escaped by preceding them with a backslash. So, if you wanted to specifically match, say 3.14, you could use "3\\.14".

You might have noticed that there's a backslash in front of the dollar amount in the step definition we're using. That's because $ itself is a metacharacter (it's an anchor, which we'll explain later), so we need to escape to make it match a normal dollar sign.

Creating more flexibility

- Character Classes
  - \\$([01234567890]*) -> or for ranges
  - \\$([0-9]*)
- Shorthand Character Classes
  - For common patterns of characters like [0-9] ->shorthand

Shorthand Character Classes

- useful shorthand character classes
  - \d stands for digit, or [0-9].
  - \w stands for word character -> [A-Za-z0-9_]
    - ***note hyphens are not included***
  - \s stands for whitespace character -> [\t\r\n]
  - \b stands for word boundary -> opposite of \w
    - Anything that is not a word character is a word boundary
  - negate shorthand character classes by capitalizing them
    - \D means any character except a digit

+

-
    - \\$(.*) * -> any number of times
        - problem any number -> means also zero times
        - so \$ without a number is also mapped -> not acceptable
- The Plus Modifier + -> also a repetition modifier
    - \\$(\\d+) -> maps number to function(int amount)

Capture the flight codes

- Capture the flight codes from all of these steps
  1. Given the flight EZY4567 is leaving today
  2. Given the flight C038 is leaving today
  3. Given a flight BA01618 is leaving today

More to capture

```
Scenario: Transfer funds from savings into checking account
    Given I have deposited $10 in my Checking Account
    And I have deposited $500 in my Savings Account
    When I transfer $500 from my Savings Account into my Checking
    Then the balance of the Checking Account should be $510
    And the balance of the Savings Account should be $0
```

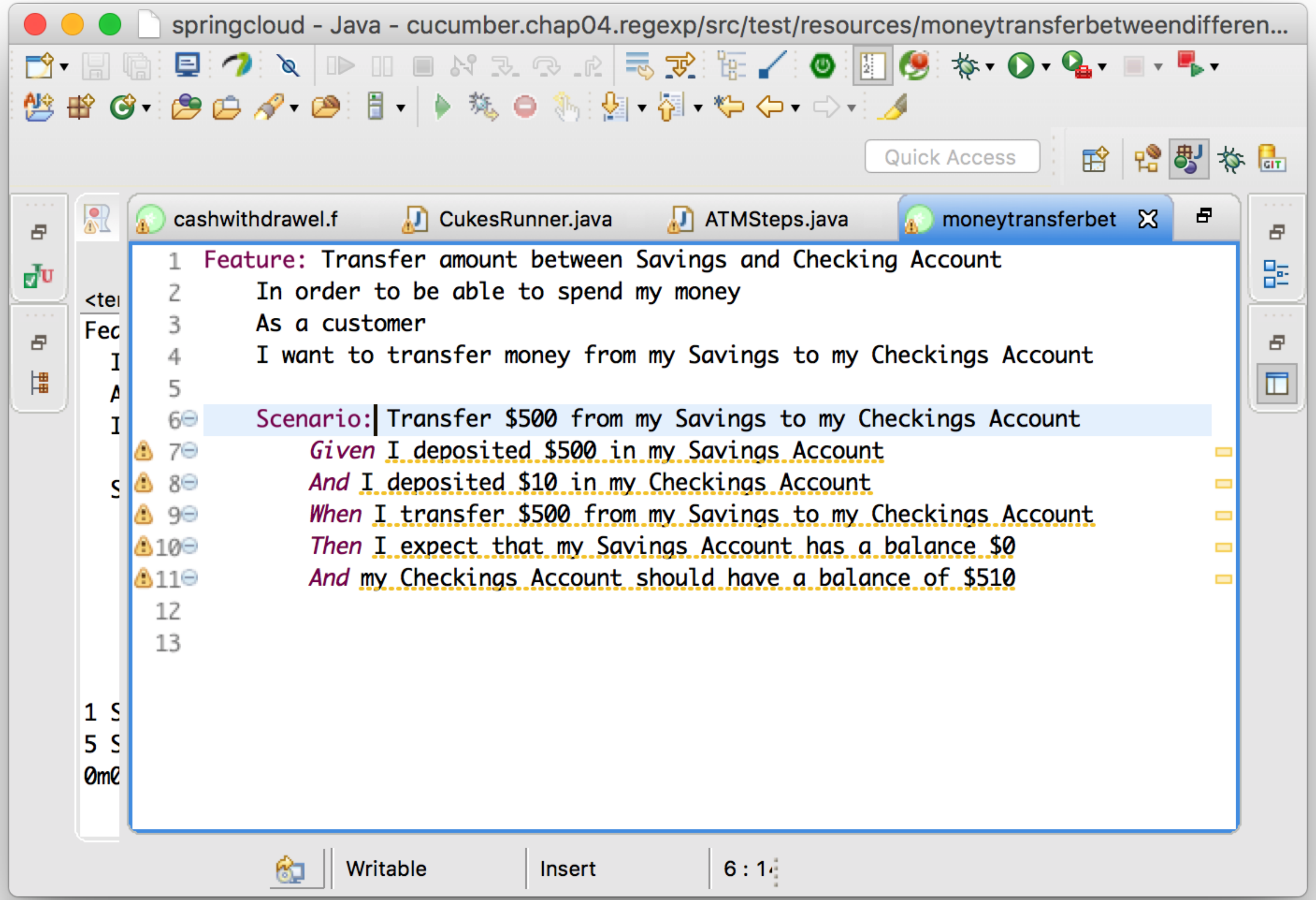
- the Given step maps to the following stepdefinition

```
@Given("I have deposited \\$(\\d+) in my (\\w+) Account")
public void iHaveDeposited$InMyAccount(int amount, String account
    // TODO: code goes here
}
```

Try this

- Write a step definition for:
  - When I transfer \$500 from my Savings Account into my Checking Account
- The step definition should capture three arguments:

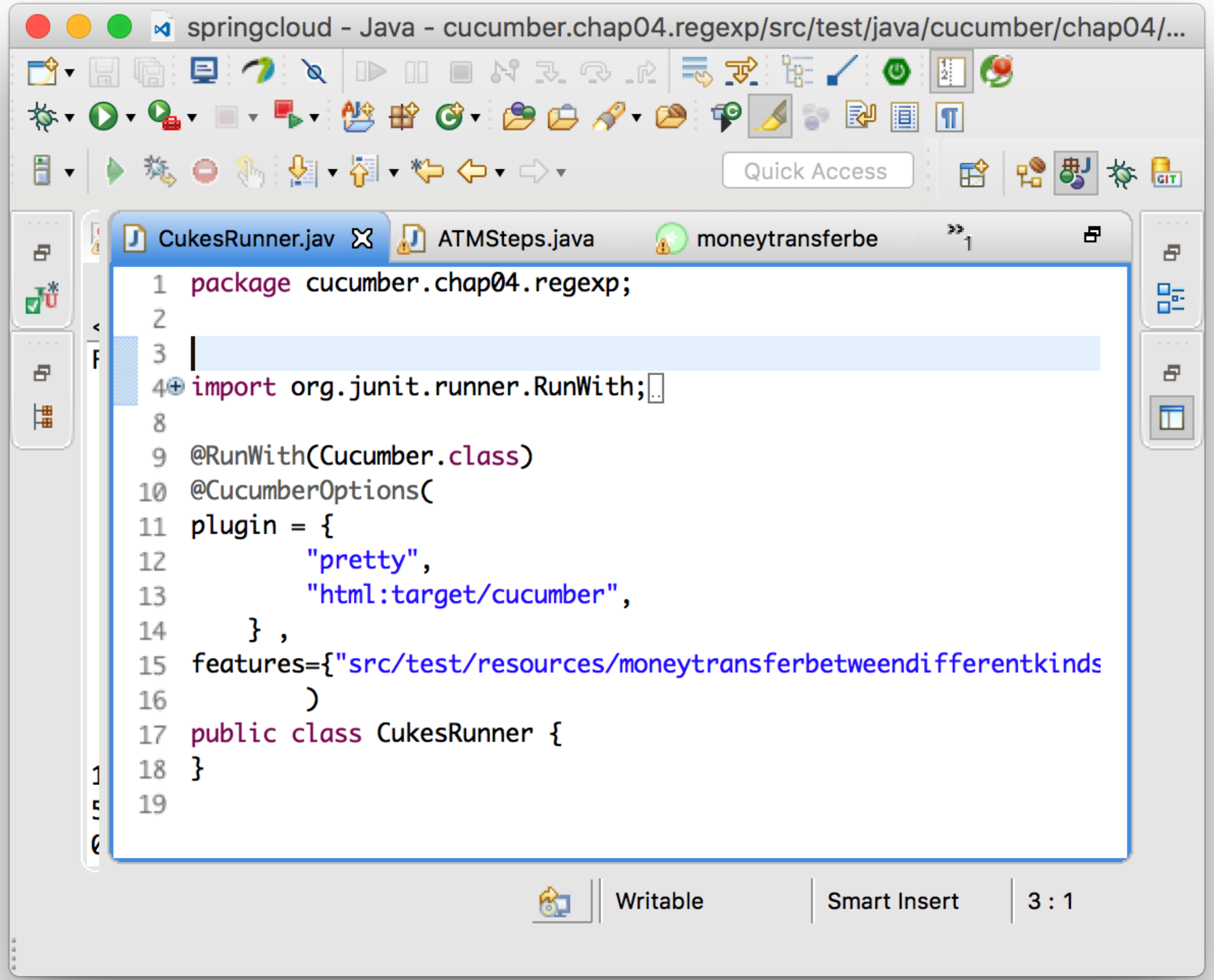
1. The amount of money being transferred
2. The type of account being debited in the transfer
3. The type of account that receives the credit in the transfer



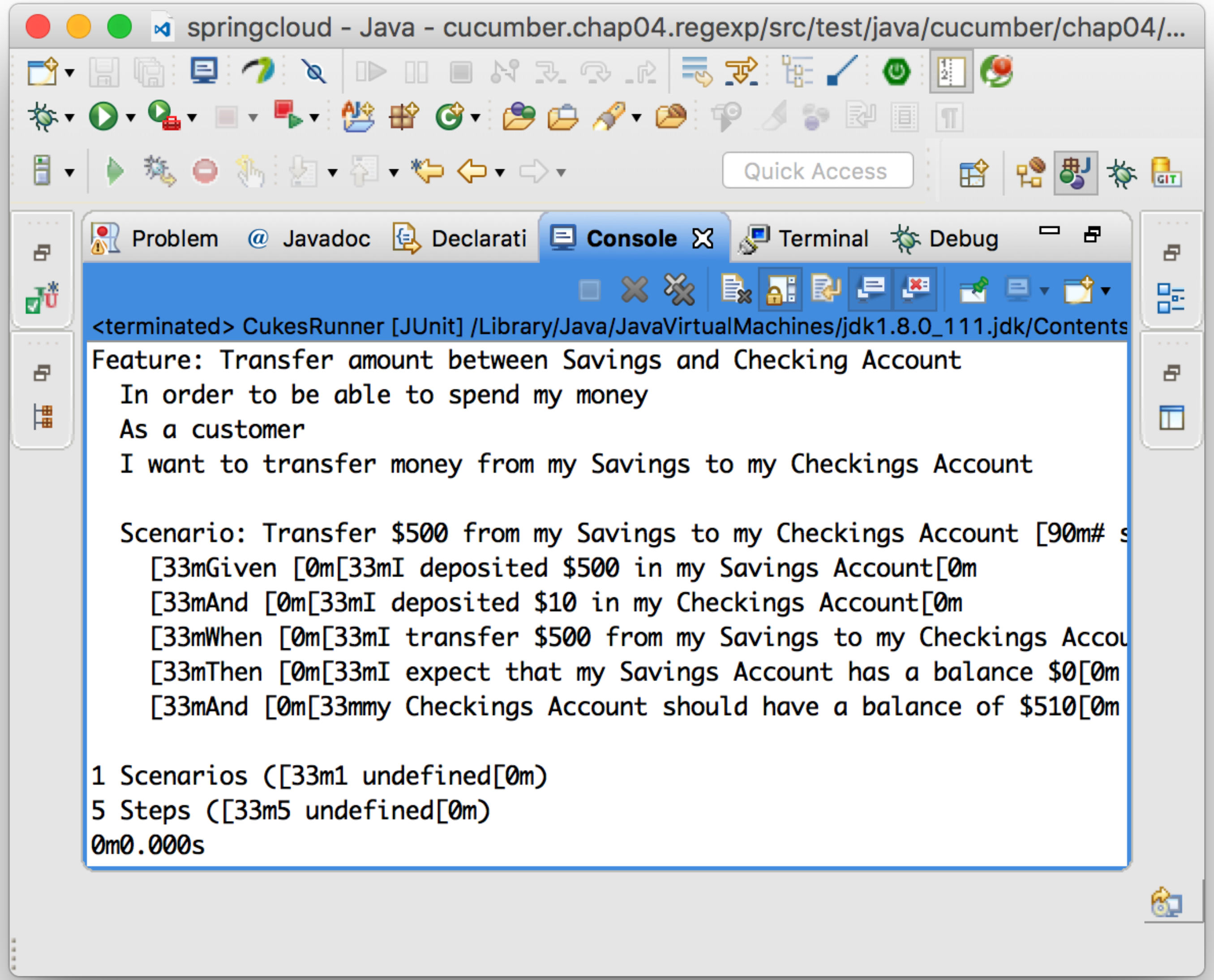
springcloud - Java - cucumber.chap04.regexp/src/test/resources/moneytransferbetweendifferen...

cashwithdrawel.f | CukesRunner.java | ATMSteps.java | moneytransferbet

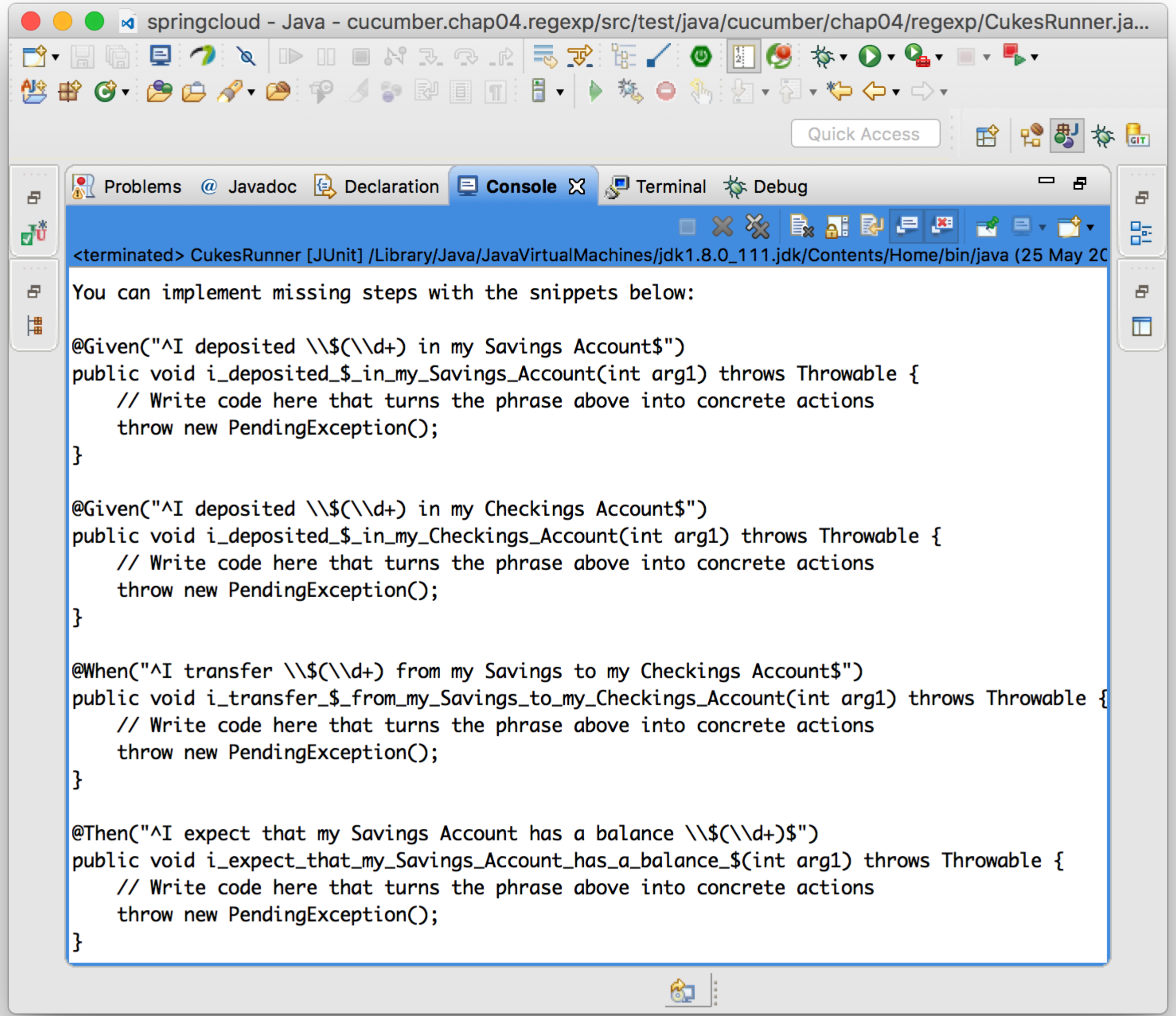
```
Feature: Transfer amount between Savings and Checking Account
    In order to be able to spend my money
    As a customer
    I want to transfer money from my Savings to my Checkings Account

    Scenario: Transfer $500 from my Savings to my Checkings Account
        Given I deposited $500 in my Savings Account
        And I deposited $10 in my Checkings Account
        When I transfer $500 from my Savings to my Checkings Account
        Then I expect that my Savings Account has a balance $0
        And my Checkings Account should have a balance of $510
```



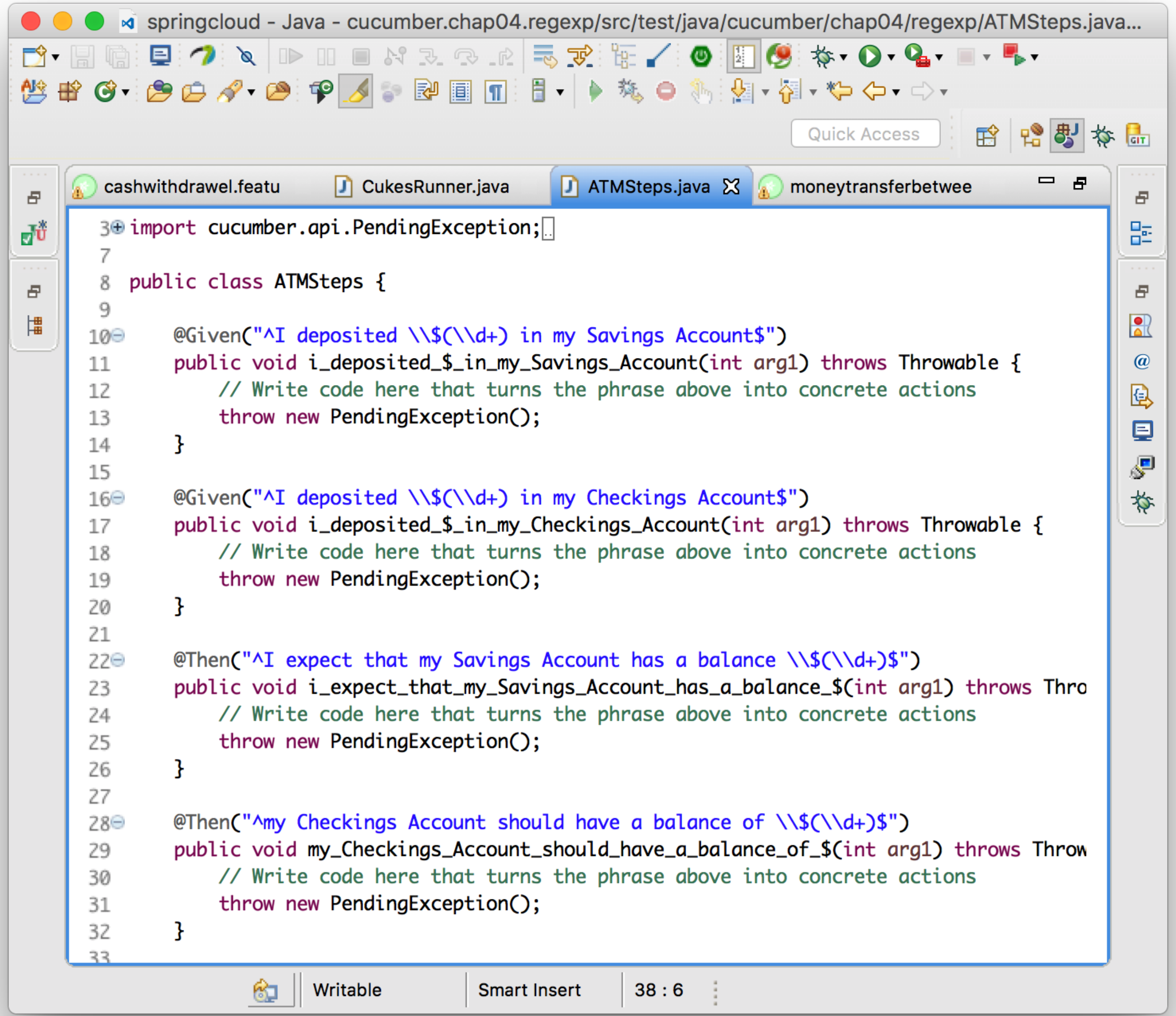
```java
package cucumber.chap04.regexp;


import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(
plugin = {
        "pretty",
        "html:target/cucumber",
    } ,
features={"src/test/resources/moneytransferbetweendifferentkinds
        )
public class CukesRunner {
}
```


springcloud - Java - cucumber.chap04.regexp/src/test/java/cucumber/chap04/...
Quick Access
Problem
Javadoc
Declarati
Console
Terminal
Debug

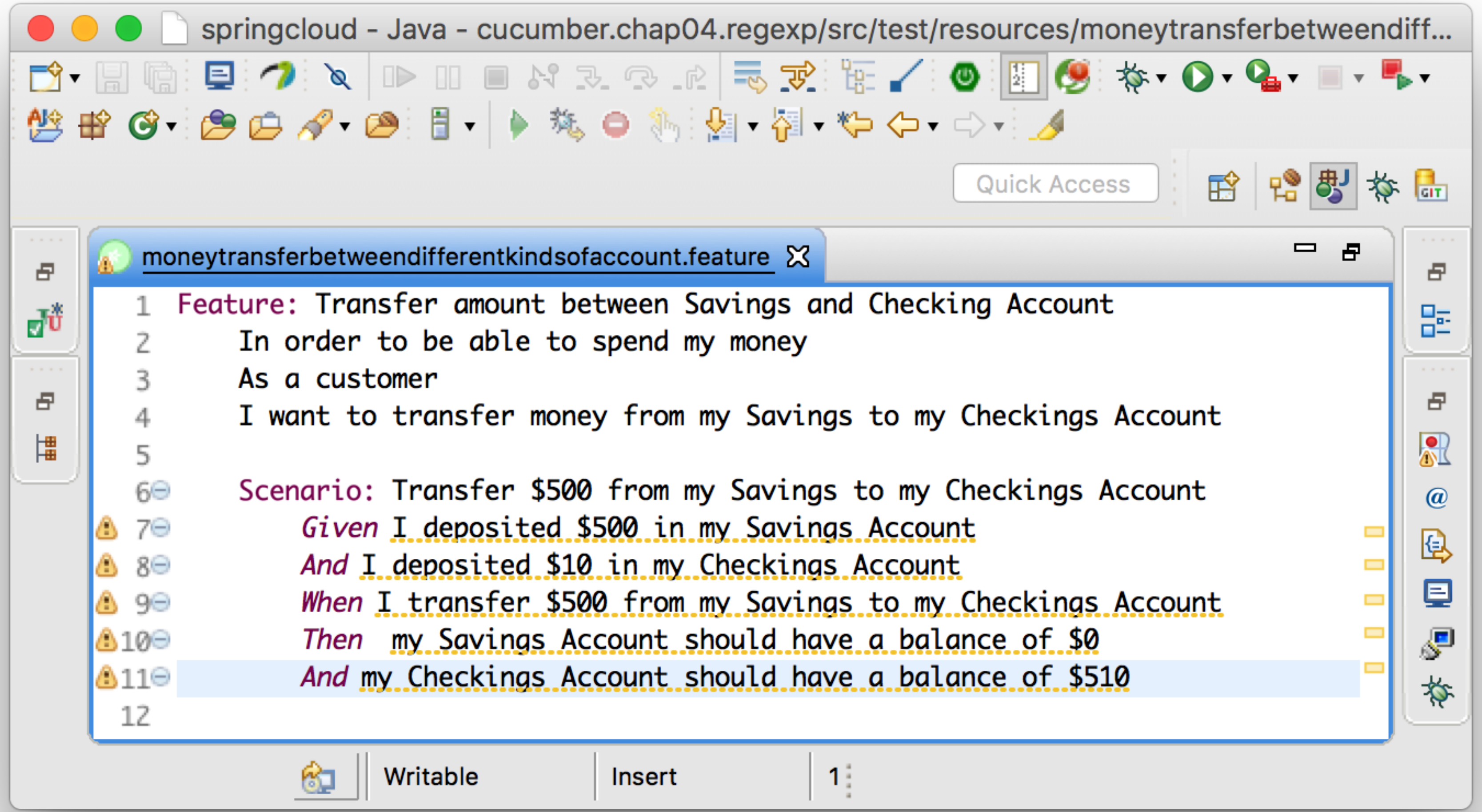
```
<terminated> CukesRunner [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents
Feature: Transfer amount between Savings and Checking Account
  In order to be able to spend my money
  As a customer
  I want to transfer money from my Savings to my Checkings Account

  Scenario: Transfer $500 from my Savings to my Checkings Account [90m# s
    [33mGiven [0m[33mI deposited $500 in my Savings Account[0m
    [33mAnd [0m[33mI deposited $10 in my Checkings Account[0m
    [33mWhen [0m[33mI transfer $500 from my Savings to my Checkings Accou
    [33mThen [0m[33mI expect that my Savings Account has a balance $0[0m
    [33mAnd [0m[33mmy Checkings Account should have a balance of $510[0m

1 Scenarios ([33m1 undefined[0m)
5 Steps ([33m5 undefined[0m)
0m0.000s
```


springcloud - Java - cucumber.chap04.regexp/src/test/java/cucumber/chap04/regexp/CukesRunner.ja...
Quick Access
Problems
Javadoc
Declaration
Console
Terminal
Debug

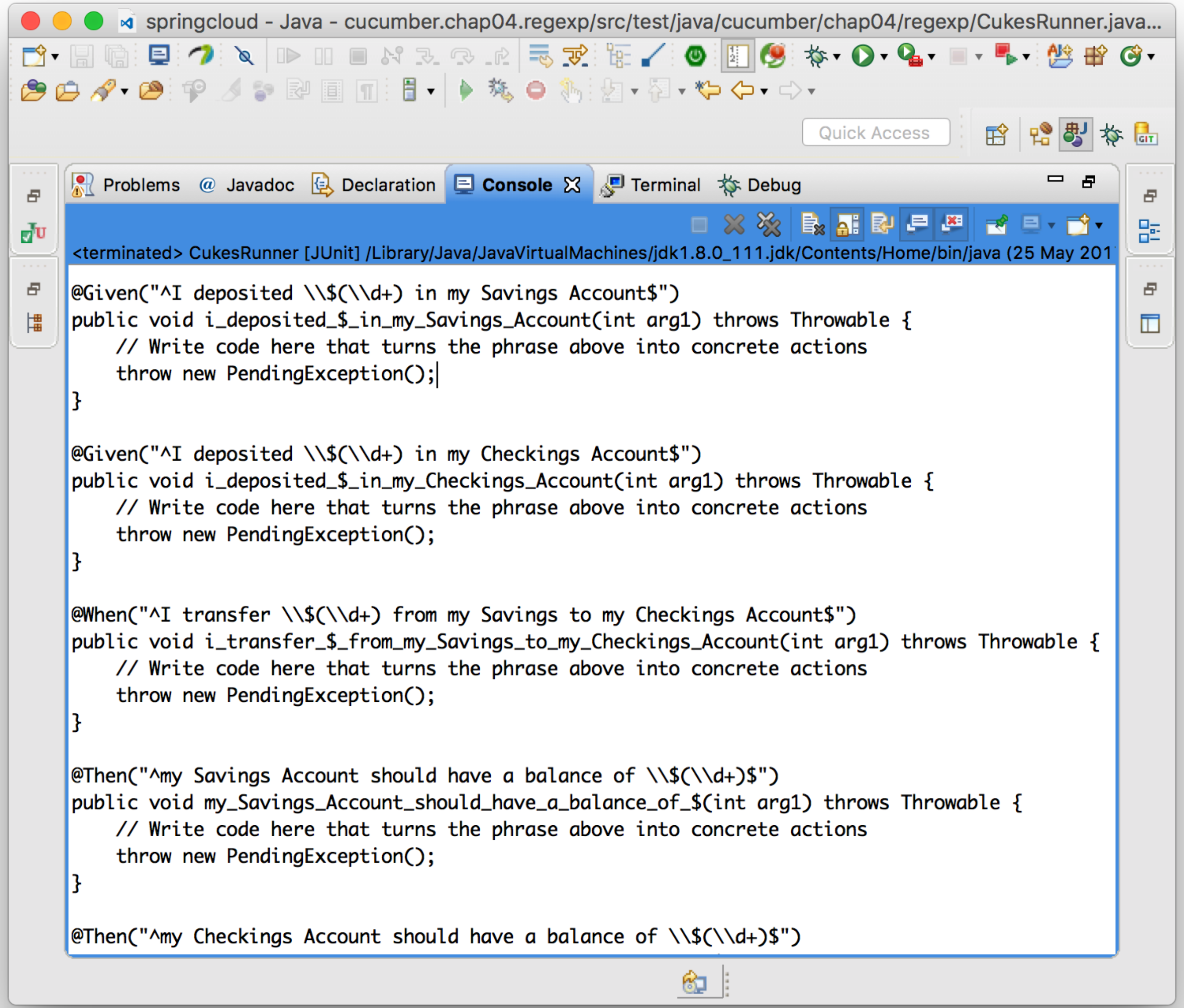
```
<terminated> CukesRunner [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java (25 May 20
You can implement missing steps with the snippets below:

@Given("^I deposited \\$(\\d+) in my Savings Account$")
public void i_deposited_$_in_my_Savings_Account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Given("^I deposited \\$(\\d+) in my Checkings Account$")
public void i_deposited_$_in_my_Checkings_Account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^I transfer \\$(\\d+) from my Savings to my Checkings Account$")
public void i_transfer_$_from_my_Savings_to_my_Checkings_Account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^I expect that my Savings Account has a balance \\$(\\d+)$")
public void i_expect_that_my_Savings_Account_has_a_balance_$(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```



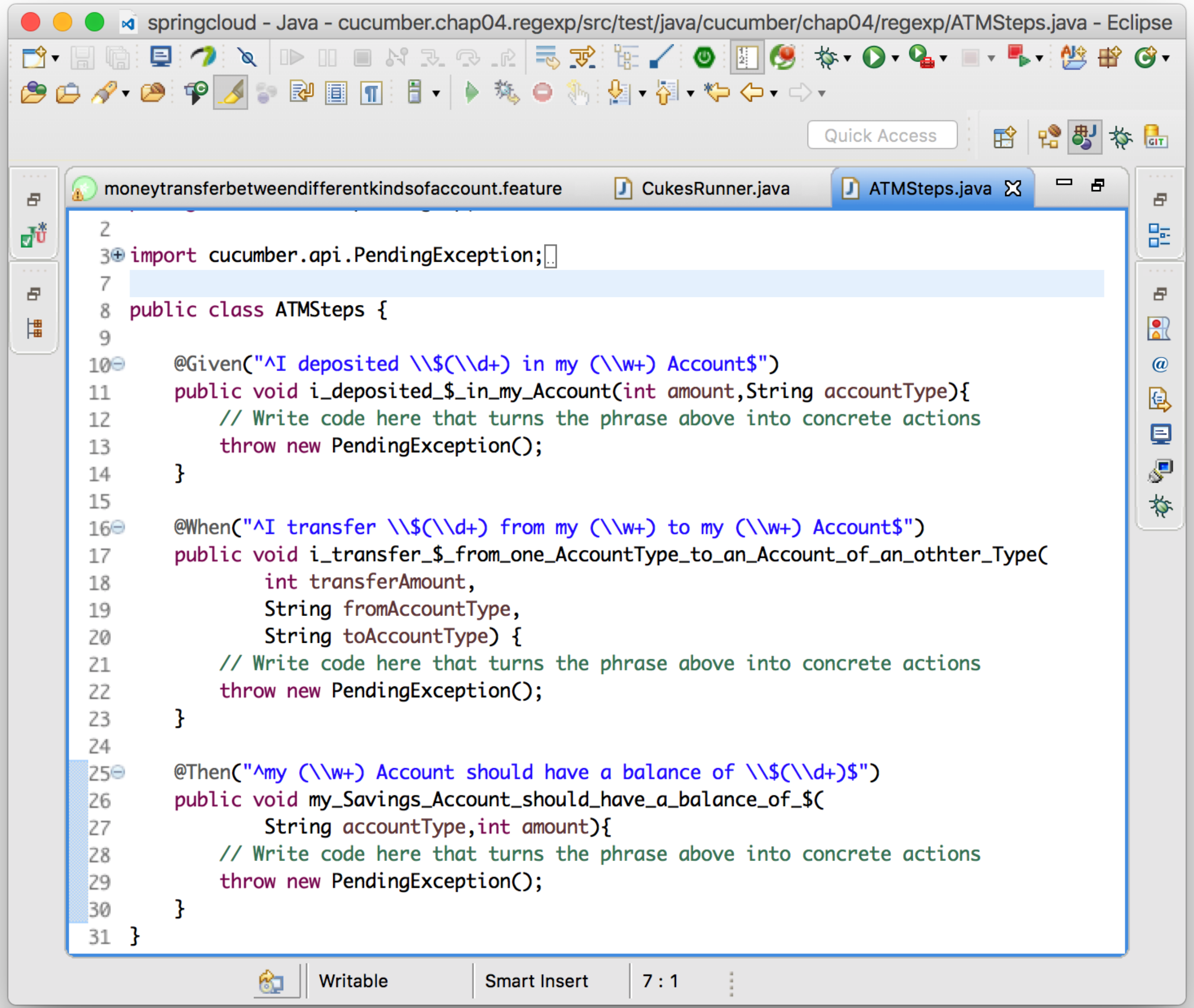
```java
import cucumber.api.PendingException;

public class ATMSteps {

    @Given("^I deposited \\$(\\d+) in my Savings Account$")
    public void i_deposited_$_in_my_Savings_Account(int arg1) throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @Given("^I deposited \\$(\\d+) in my Checkings Account$")
    public void i_deposited_$_in_my_Checkings_Account(int arg1) throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @Then("^I expect that my Savings Account has a balance \\$(\\d+)$")
    public void i_expect_that_my_Savings_Account_has_a_balance_$(int arg1) throws Thro
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @Then("^my Checkings Account should have a balance of \\$(\\d+)$")
    public void my_Checkings_Account_should_have_a_balance_of_$(int arg1) throws Throw
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }
```


springcloud - Java - cucumber.chap04.regexp/src/test/resources/moneytransferbetweendiff...
Quick Access
moneytransferbetweendifferentkindsofaccount.feature
Writable
Insert
1

```
Feature: Transfer amount between Savings and Checking Account
    In order to be able to spend my money
    As a customer
    I want to transfer money from my Savings to my Checkings Account

    Scenario: Transfer $500 from my Savings to my Checkings Account
        Given I deposited $500 in my Savings Account
        And I deposited $10 in my Checkings Account
        When I transfer $500 from my Savings to my Checkings Account
        Then  my Savings Account should have a balance of $0
        And my Checkings Account should have a balance of $510
```


springcloud - Java - cucumber.chap04.regexp/src/test/java/cucumber/chap04/regexp/CukesRunner.java...
Quick Access
Problems
Javadoc
Declaration
Console
Terminal
Debug

```
<terminated> CukesRunner [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java (25 May 201

@Given("^I deposited \\$(\\d+) in my Savings Account$")
public void i_deposited_$_in_my_Savings_Account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Given("^I deposited \\$(\\d+) in my Checkings Account$")
public void i_deposited_$_in_my_Checkings_Account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^I transfer \\$(\\d+) from my Savings to my Checkings Account$")
public void i_transfer_$_from_my_Savings_to_my_Checkings_Account(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^my Savings Account should have a balance of \\$(\\d+)$")
public void my_Savings_Account_should_have_a_balance_of_$(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^my Checkings Account should have a balance of \\$(\\d+)$")
```



springcloud - Java - cucumber.chap04.regexp/src/test/java/cucumber/chap04/regexp/ATMSteps.java - Eclipse

moneytransferbetweendifferentkindsofaccount.feature | CukesRunner.java | ATMSteps.java

```java
import cucumber.api.PendingException;

public class ATMSteps {

    @Given("^I deposited \\$(\\d+) in my (\\w+) Account$")
    public void i_deposited_$_in_my_Account(int amount,String accountType){
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @When("^I transfer \\$(\\d+) from my (\\w+) to my (\\w+) Account$")
    public void i_transfer_$_from_one_AccountType_to_an_Account_of_an_othter_Type(
            int transferAmount,
            String fromAccountType,
            String toAccountType) {
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @Then("^my (\\w+) Account should have a balance of \\$(\\d+)$")
    public void my_Savings_Account_should_have_a_balance_of_$(
            String accountType,int amount){
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }
}
```

Writable | Smart Insert | 7 : 1

Readability

- readability of features -> teams learn ubiquitous language
- a really benefit -> consistent use of terminology reduces misunderstanding
- allows a more smoothly communication in the team
- encourage feature authors consistent use of nouns and verbs
  - on the other hand enable authors to express themselves as naturally as possible

Keep features readable and natural

- develop skills to make step definitions flexible enough to match the different ways something might be expressed by a feature author

The Question Mark Modifier

```
Given I have 1 cucumber in my basket
Given I have 256 cucumbers in my basket
```

- The question mark makes the preceding character optional

```
@Given("I have (\\d+) cucumbers? in my basket")
public void iHaveCucumbersInMyBasket(int number) {
// TODO: code goes here
}
```

Noncapturing Groups

- add flexibility to our step definitions: letting feature authors say the same thing in slightly different ways

```
When I visit the homepage

When I go to the homepage
```

```
@When("I (?:visit|go to) the homepage")
public void iVisitTheHomepage() {
// TODO: code goes here
}
```

- The ?: at the start of the group marks it as noncapturing, meaning Cucumber won’t pass it as an argument to our block

Anchors

- the ^ and $ are two metacharacters called anchors -> they tie down each end, the start and end of the string
- omit one or both of them -> results in a much more flexible step definition —> perhaps too flexible

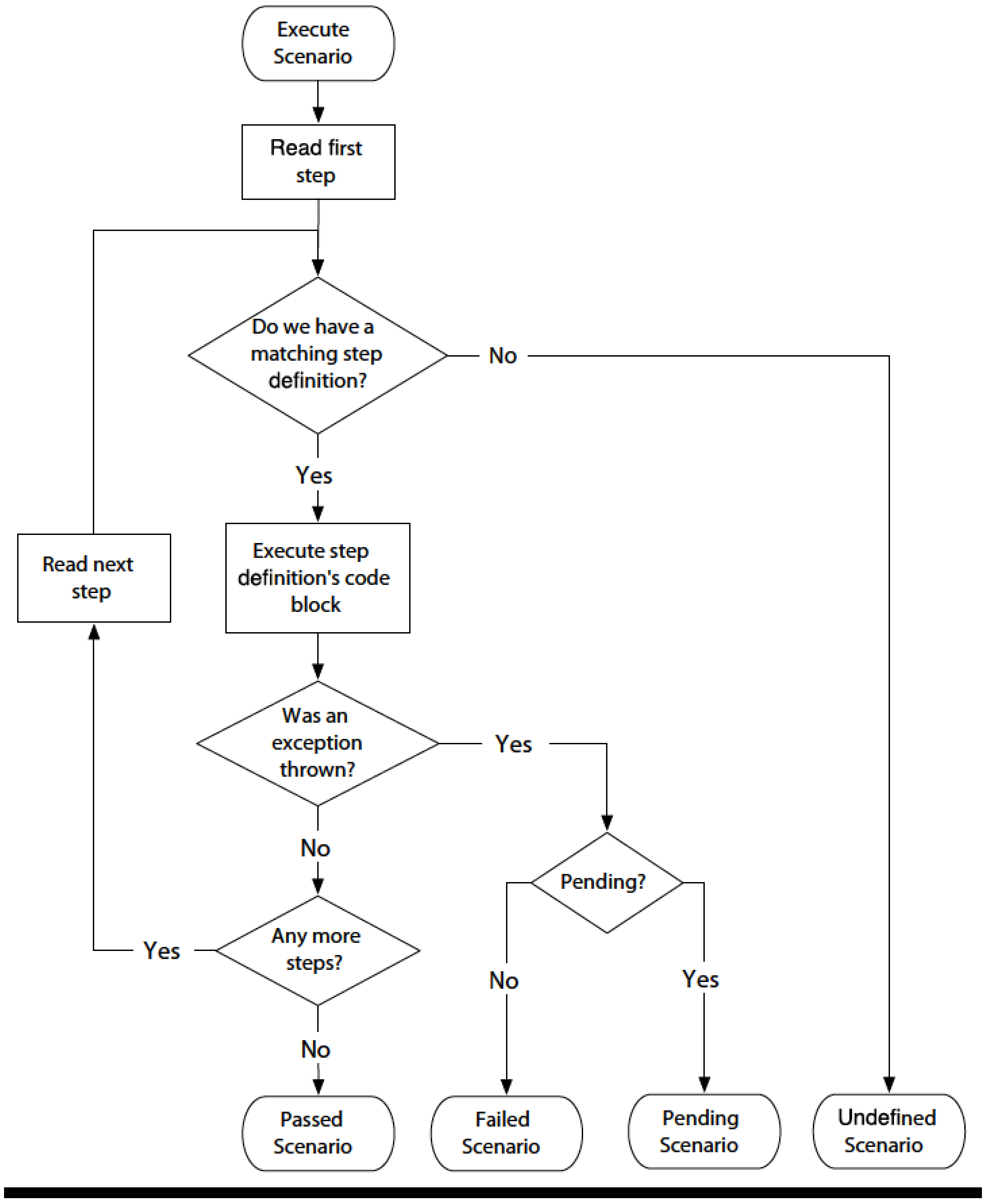
```
@Given("^I have deposited \\$(\\d+) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
// TODO: code goes here
}
```

- This matches

```
Given I have deposited $100 in my Account from a check my Grandma
```

Returning Results

- Cucumber is a testing tool
- The Java code of a step definition our tests find out whether a step has succeeded
- Cucumber uses exceptions to communicate the failure of a test


Execute Scenario
Read first step
Do we have a matching step definition?
No
Yes
Execute step definition's code block
Read next step
Was an exception thrown?
Yes
No
Pending?
Any more steps?
Yes
No
No
Yes
Passed Scenario
Failed Scenario
Pending Scenario
Undefined Scenario

Assertions and Exceptions

Even if you're used to using a testing library like JUnit, you might not have realized that the assertions in those libraries work by raising exceptions.

You can prove this to yourself by writing a little Java program that runs a failing assertion:

step_definitions/assertions_sidebar/AssertionExample.java

```java
import org.junit.*;
import static org.junit.Assert.*;

public class AssertionExample {

  public static void main(String[] args) {
    try {
      assertTrue(false);
    } catch (AssertionError e) {
      System.out.print("Exception was raised was ");
      System.out.println(e.getClass().getName());
    }
  }
}
```

When you run it, you should find that this program raises an exception of type java.lang.AssertionError.

Strict Mode

If you use the --strict command-line option in your shell script, ./cucumber, then it will return an exit code of 1 (to indicate an error) if there are any undefined or pending steps.

This can be useful in a continuous integration build to spot any half-finished features that have been accidentally checked in or when you've refactored your step definitions and some of your steps are no longer matching.

Cucumber

- assumes that a step has passed unless its step definition throws an exception
- If the exception thrown is a PendingException
  - then the step is marked as pending
  - all other exceptions cause the step to fail
- If a step passes, Cucumber moves on to the next step

Undefined Steps

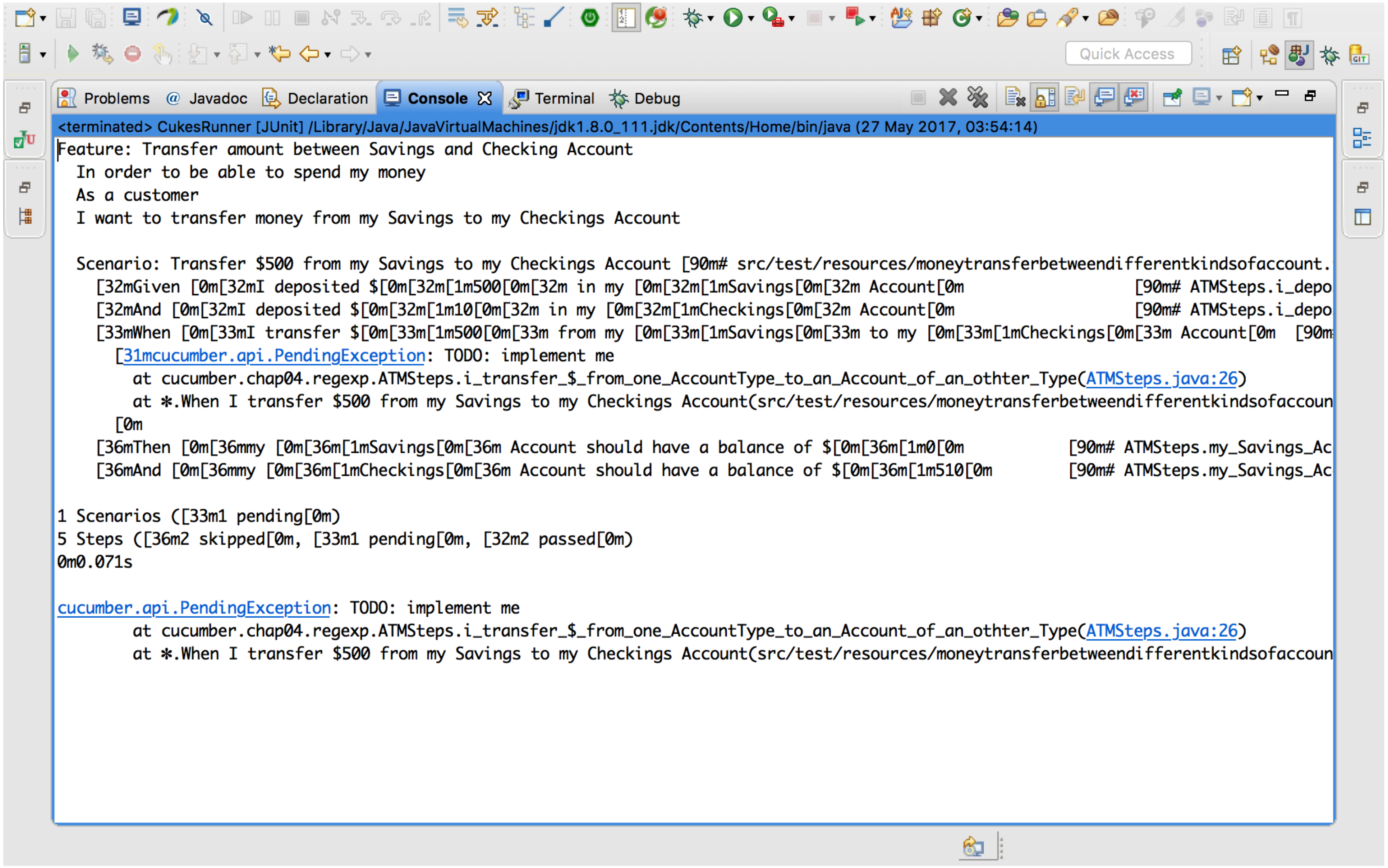
- When no step definition matches a step -> then step is marked as undefined (yellow) and stops the scenario
- The rest of the steps will be either skipped or marked as undefined

Pending Steps

- A step definition that’s halfway through being implemented is marked as pending
- the scenario will be stopped
- throwing a PendingException tells the Cucumber’s runtime that the step has failed
  - -> in a particular way: the step definition is still being worked on

Failing Steps

- step definition will fail for one of two reasons
- The scenario couldn’t finish because you have a bug in your step definition code, or in the system under test
- The step definition has used an assertion to check something about the state of the system, and the check didn’t pass.



```
<terminated> CukesRunner [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java (27 May 2017, 03:54:14)
Feature: Transfer amount between Savings and Checking Account
  In order to be able to spend my money
  As a customer
  I want to transfer money from my Savings to my Checkings Account

  Scenario: Transfer $500 from my Savings to my Checkings Account [90m# src/test/resources/moneytransferbetweendifferentkindsofaccount.
    [32mGiven [0m[32mI deposited $[0m[32m[1m500[0m[32m in my [0m[32m[1mSavings[0m[32m Account[0m                  [90m# ATMSteps.i_depo
    [32mAnd [0m[32mI deposited $[0m[32m[1m10[0m[32m in my [0m[32m[1mChecking[0m[32m Account[0m                   [90m# ATMSteps.i_depo
    [33mWhen [0m[33mI transfer $[0m[33m[1m500[0m[33m from my [0m[33m[1mSavings[0m[33m to my [0m[33m[1mCheckings[0m[33m Account[0m  [90m
      [31mcucumber.api.PendingException: TODO: implement me
        at cucumber.chap04.regexp.ATMSteps.i_transfer_$_from_one_AccountType_to_an_Account_of_an_othter_Type(ATMSteps.java:26)
        at ✱.When I transfer $500 from my Savings to my Checkings Account(src/test/resources/moneytransferbetweendifferentkindsofaccoun
      [0m
    [36mThen [0m[36mmy [0m[36m[1mSavings[0m[36m Account should have a balance of $[0m[36m[1m0[0m          [90m# ATMSteps.my_Savings_Ac
    [36mAnd [0m[36mmy [0m[36m[1mChecking[0m[36m Account should have a balance of $[0m[36m[1m510[0m        [90m# ATMSteps.my_Savings_Ac

1 Scenarios ([33m1 pending[0m)
5 Steps ([36m2 skipped[0m, [33m1 pending[0m, [32m2 passed[0m)
0m0.071s

cucumber.api.PendingException: TODO: implement me
        at cucumber.chap04.regexp.ATMSteps.i_transfer_$_from_one_AccountType_to_an_Account_of_an_othter_Type(ATMSteps.java:26)
        at ✱.When I transfer $500 from my Savings to my Checkings Account(src/test/resources/moneytransferbetweendifferentkindsofaccoun
```

What We Just Learned

- Think of a step definition as being a special kind of method
  - It can be invoked by any step that matches its regular expression
- Regular expressions can contain wildcards -> flexibility to make the Gherkin steps nice and readable
- Keep Java step definition code clean and free of duplication

About Step definitions

- map from the Gherkin scenarios’ plainlanguage descriptions of user actions into Java code, which simulates those actions
- registered with Cucumber by using @Given, @When, @Then, or one of the aliases for a spoken language
- use regular expressions to declare the steps that they can handle -> regular expressions can contain wildcards, one step definition can handle several different steps
- communicates its result to Cucumber by raising, or not raising, an exception

Expressive Scenarios

- When writing Cucumber features -> readability is the main goal
- Key to expressive scenarios is having a healthy vocabulary of domain language to use to express your requirements
- Need more then only the basic set of Gherkin keywords -> otherwise the scenarios become boring
- Remove repetitive clutter by using special Gherkin syntax

Background

- A background section in a feature file allows -> to specify a set of steps that are common to every scenario in the file
- If you ever need to change those steps -> change them in only one place
- The importance of those steps fades into the background -> when reading each individual scenario, you can focus on what is unique and important about that scenario

Background example

```
Feature: Change PIN

    Scenario: Change PIN successfully
        Given I have been issued a new card
        And I insert the card, entering the correct PIN
        When I choose "Change PIN" from the menu
        And I change the PIN to 9876
        Then the system should remember my PIN is now 9876

    Scenario: Try to change PIN to the same as before
        Given I have been issued a new card
        And I insert the card, entering the correct PIN
        When I choose "Change PIN" from the menu
        And I try to change the PIN to the original PIN number
        Then I should see a warning message
```

Discussion

- The first 3 steps in each scenario, while necessary to clarify the context of the scenario, are completely repeated in both scenarios
- -> distracts, makes it harder to see the essence of what each scenario is testing
- factor out the 3 repeated steps into a Background

Rewrite with background

```
Feature: Change PIN

    Background:
        Given I have been issued a new card
        And I insert the card, entering the correct PIN
        And I choose "Change PIN" from the menu

    Scenario: Change PIN successfully
        When I change the PIN to 9876
        Then the system should remember my PIN is now 9876

    Scenario: Try to change PIN to the same as before
        When I try to change the PIN to the original PIN number
        Then I should see a warning message
        And the system should not have changed my PIN
```

Background tips

Refactoring to Background

Refactoring[a] is the process of changing code to improve its readability or design without changing its behavior. This technique applies to Gherkin features just as well as it does to the rest of your codebase. As your understanding of your domain grows through the course of the project, you'll want to reflect that learning by updating your features.

Often you don't see a background immediately. You might start out by writing one or two scenarios, and it's only as you write the third that you notice some common steps. When you spot a feature where the same or similar steps are repeated in several scenarios, see whether you can refactor to extract those steps into a background. It can take a little bit of courage to do this, because there's a risk you might make a mistake and break something, but this is a pretty safe refactoring. Once you're done, you should end up with the feature doing exactly the same thing as it did before you started but easier to read.

a. *Refactoring: Improving the Design of Existing Code [FBBO99]*

Data tables

- When a lot of data is involved

```
Given a User "Michael Jackson" born on August 29, 1958
And a User "Elvis" born on January 8, 1935
And a User "John Lennon" born on October 9, 1940

Given these Users:
| name                | date of birth    |
| Michael Jackson     | August 29, 1958 |
| Elvis               | January 8, 1935 |
| John Lennon         | October 9, 1940 |
```

About data tables

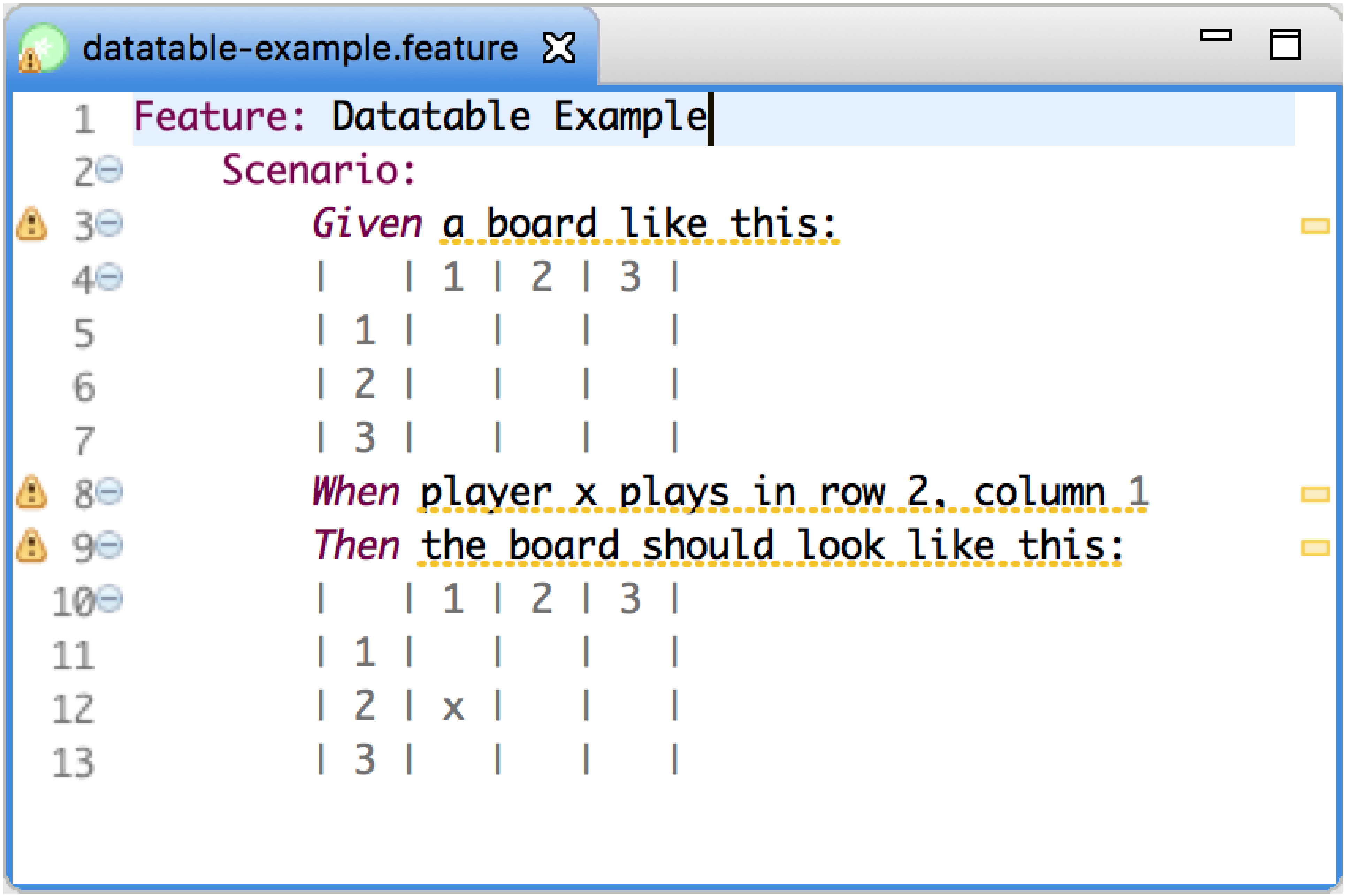
- table starts on the line immediately following the step
- cells are separated using the pipe character: |
- line up the pipes using whitespace
  - Cucumber ignores the surrounding whitespace
- freedom to specify data in different ways

Freedom in defining data tables

```
Then I should see a vehicle that matches the following descriptio
| Wheels          | 2                            |
| Max Speed       | 60 mph                       |
| Accessories     | lights, shopping basket      |

Or just to specify a list:
Then my shopping list should contain:
| Onions      |
| Potatoes    |
| Sausages    |
| Apples      |
| Relish      |
```

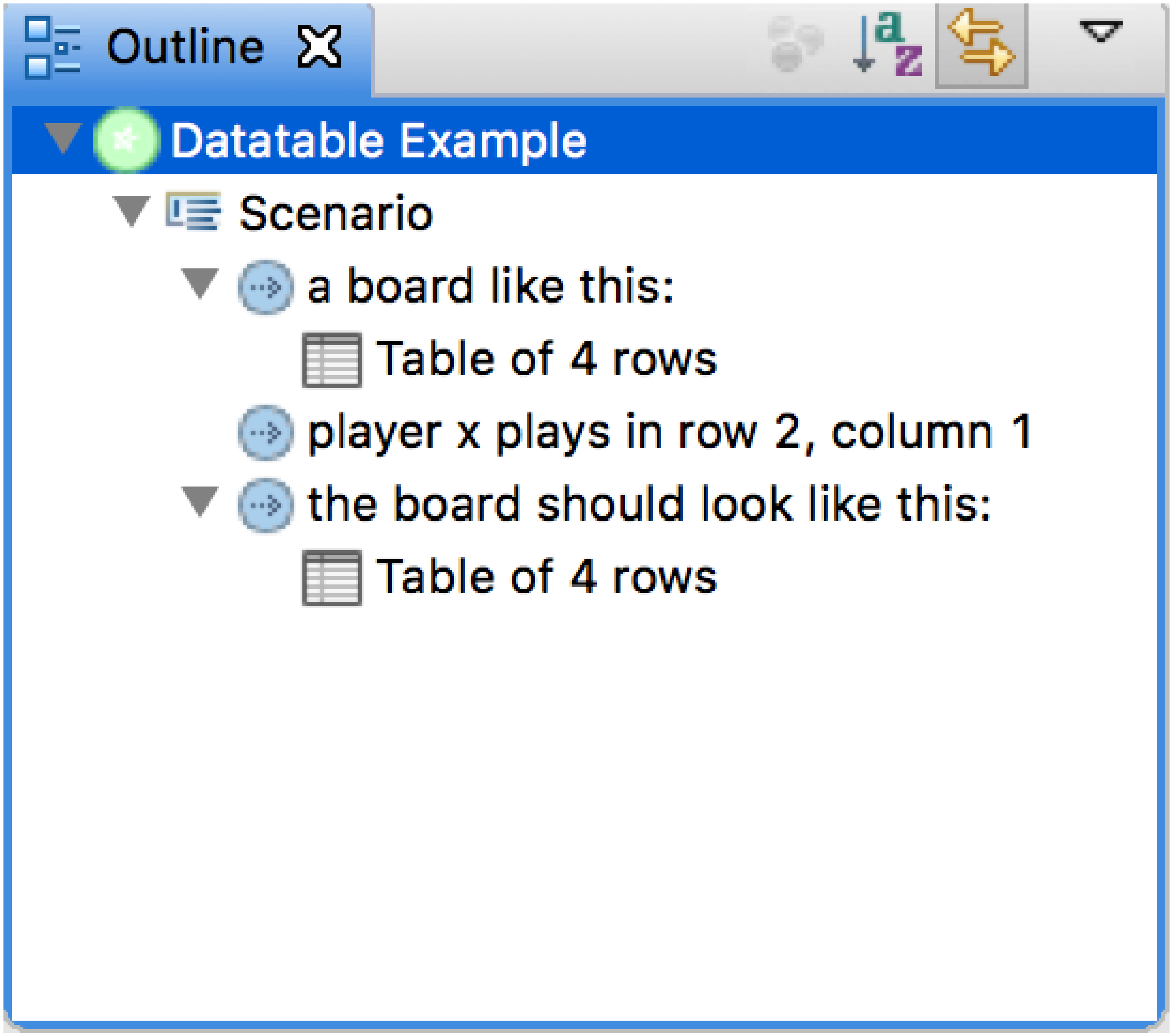
Working with Data Tables in Step Definitions

```
Feature:
    Scenario:
        Given a board like this:
        |   | 1 | 2 | 3 |
        | 1 |   |   |   |
        | 2 |   |   |   |
        | 3 |   |   |   |
        When player x plays in row 2, column 1
        Then the board should look like this:
        |   | 1 | 2 | 3 |
        | 1 |   |   |   |
        | 2 | x |   |   |
        | 3 |   |   |   |
```

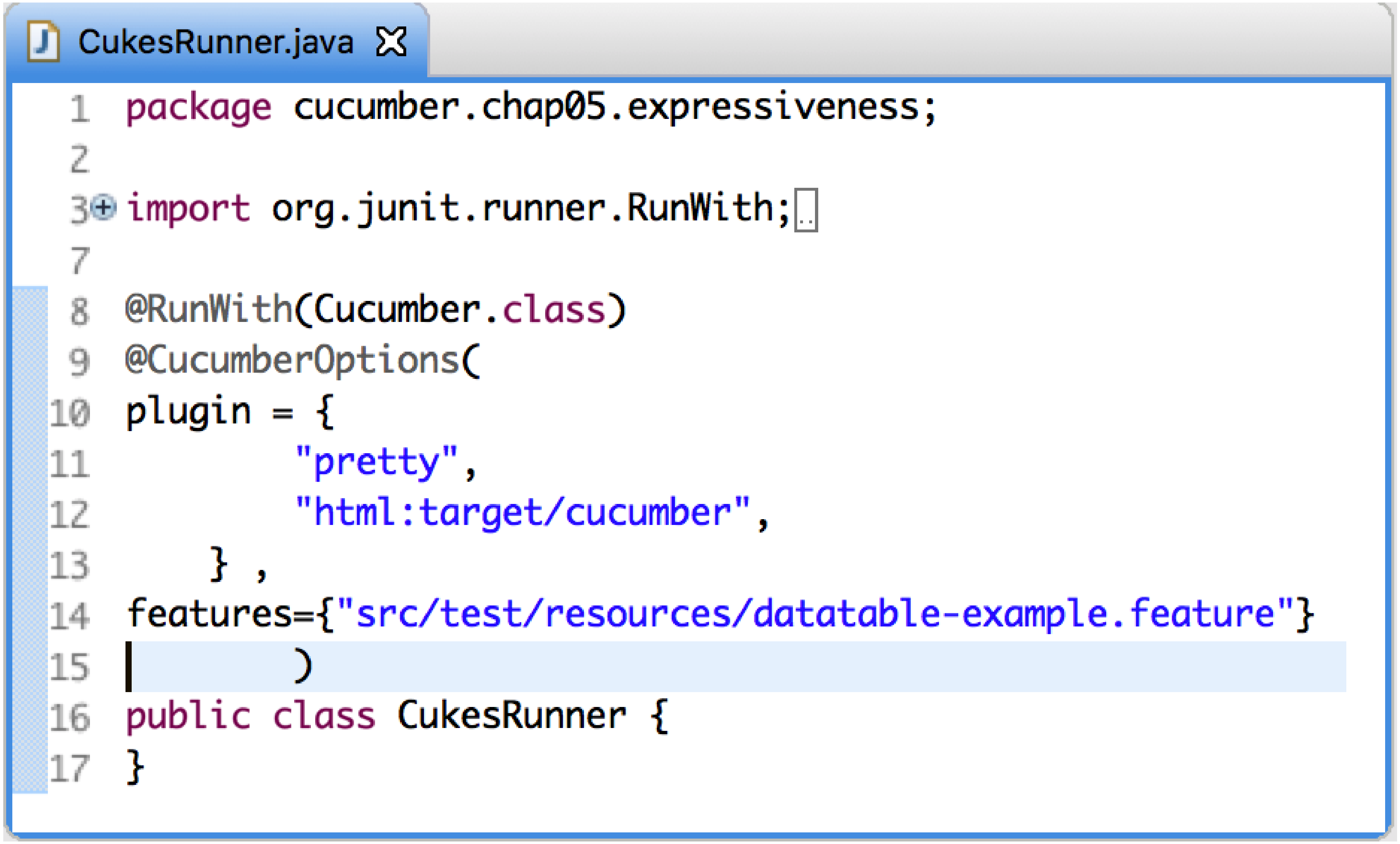


datatable-example.feature

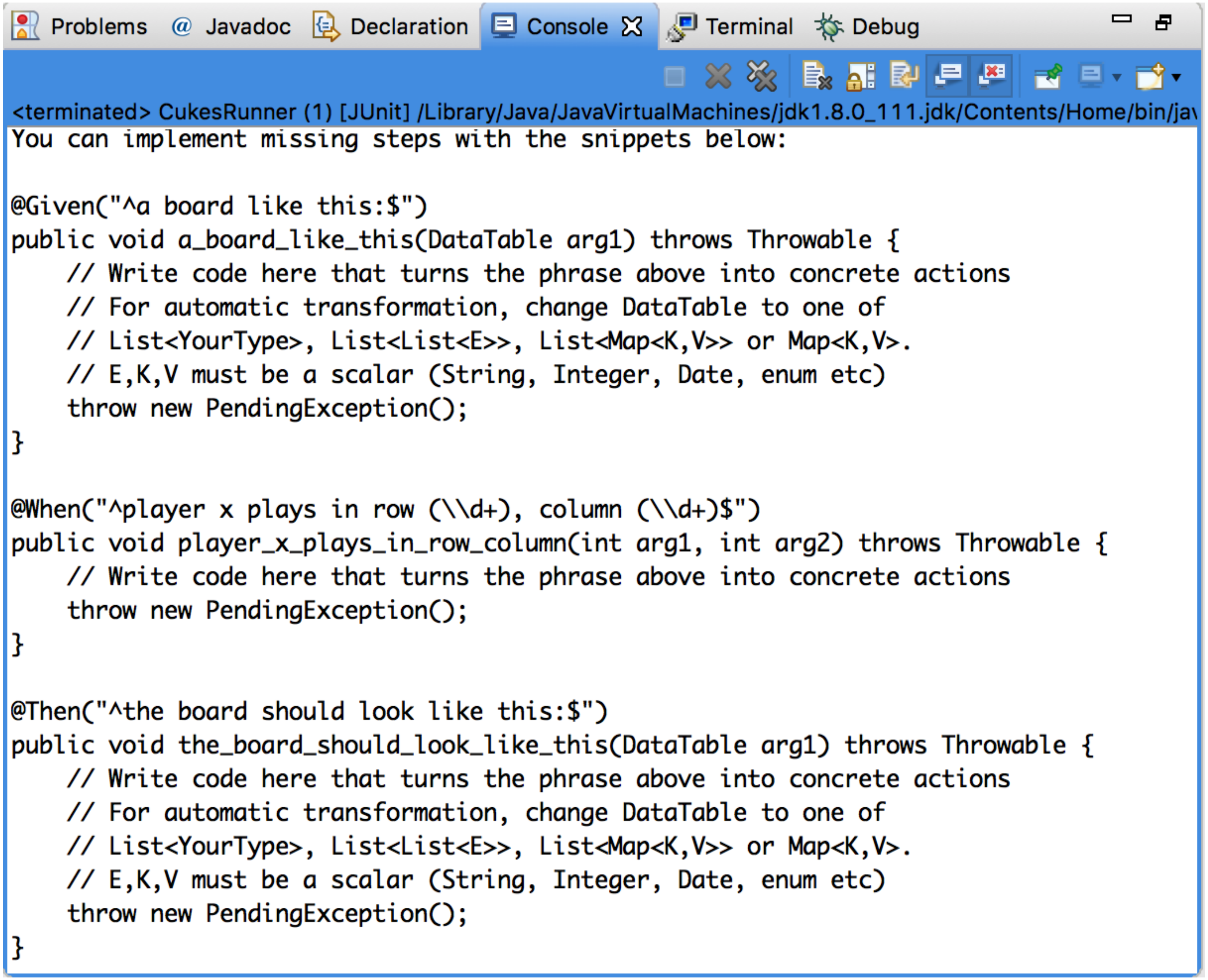
```
Feature: Datatable Example
    Scenario:
        Given a board like this:
        |   | 1 | 2 | 3 |
        | 1 |   |   |   |
        | 2 |   |   |   |
        | 3 |   |   |   |
        When player x plays in row 2, column 1
        Then the board should look like this:
        |   | 1 | 2 | 3 |
        | 1 |   |   |   |
        | 2 | x |   |   |
        | 3 |   |   |   |
```



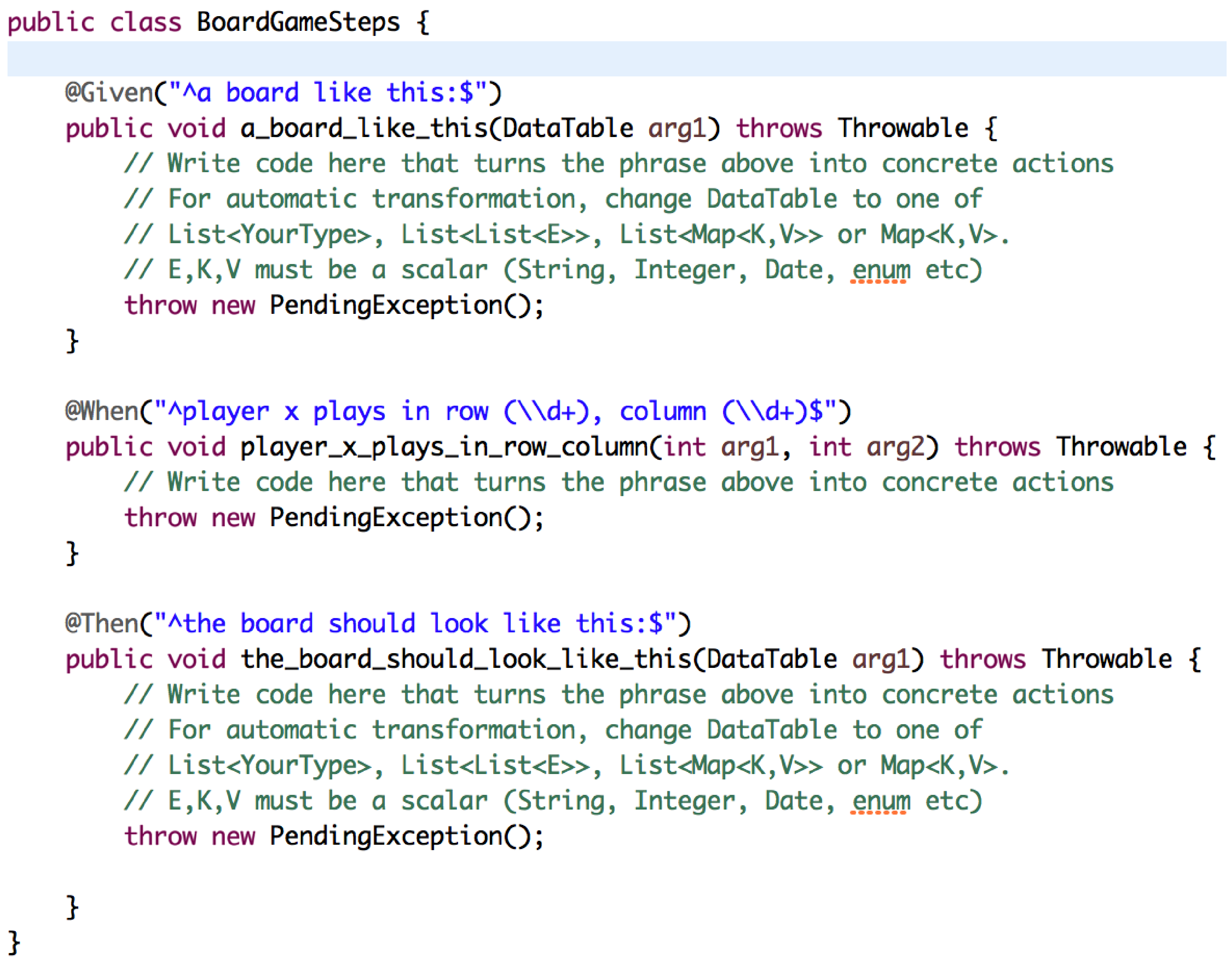
Datatable Example
Scenario
a board like this:
Table of 4 rows
player x plays in row 2, column 1
the board should look like this:
Table of 4 rows


CukesRunner.java

```
package cucumber.chap05.expressiveness;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(
plugin = {
        "pretty",
        "html:target/cucumber",
    } ,
features={"src/test/resources/datatable-example.feature"}
        )
public class CukesRunner {
}
```



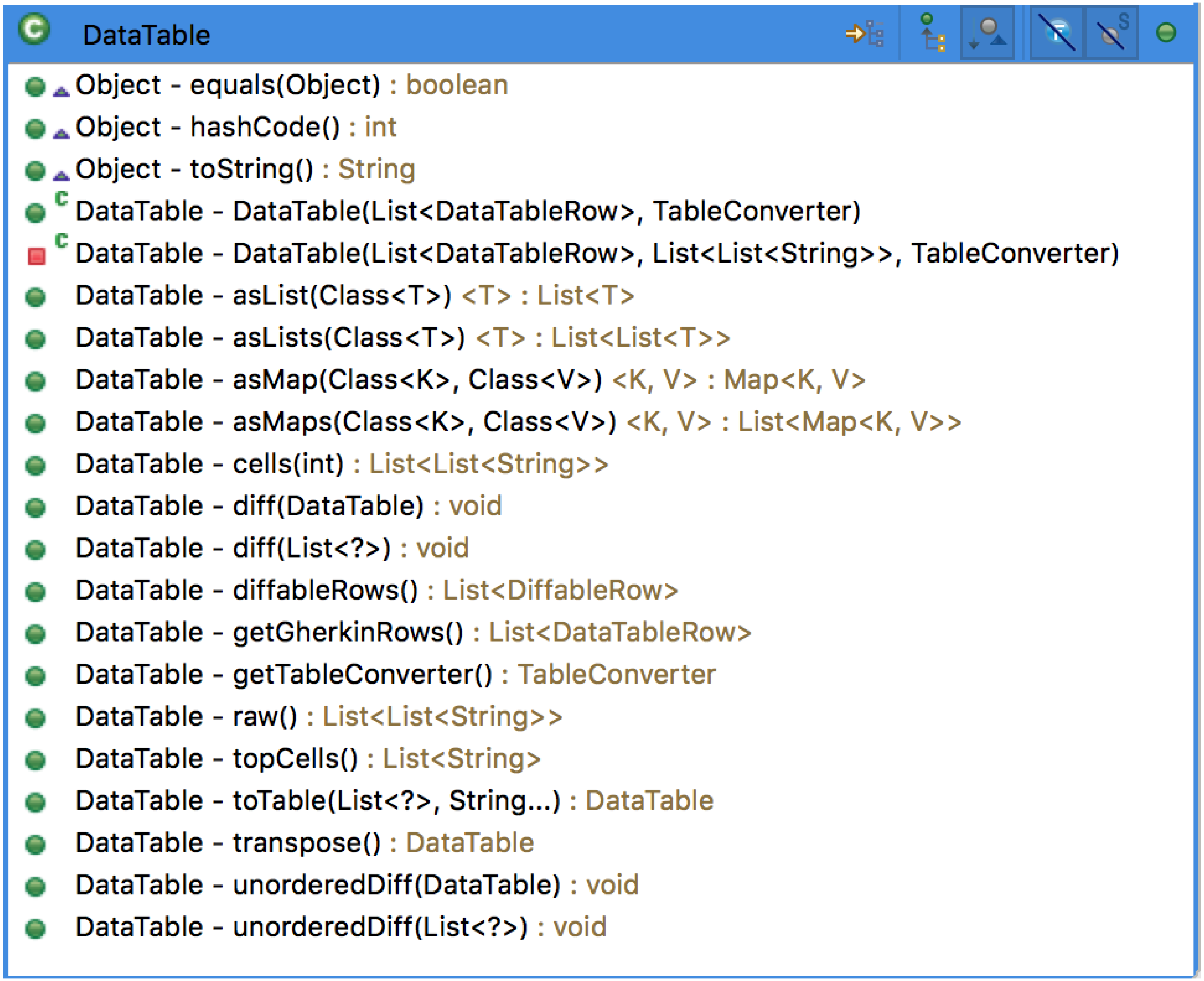
```
<terminated> CukesRunner (1) [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/jav
You can implement missing steps with the snippets below:

@Given("^a board like this:$")
public void a_board_like_this(DataTable arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    // For automatic transformation, change DataTable to one of
    // List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V>.
    // E,K,V must be a scalar (String, Integer, Date, enum etc)
    throw new PendingException();
}

@When("^player x plays in row (\\d+), column (\\d+)$")
public void player_x_plays_in_row_column(int arg1, int arg2) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^the board should look like this:$")
public void the_board_should_look_like_this(DataTable arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    // For automatic transformation, change DataTable to one of
    // List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V>.
    // E,K,V must be a scalar (String, Integer, Date, enum etc)
    throw new PendingException();
}
```



```java
public class BoardGameSteps {

    @Given("^a board like this:$")
    public void a_board_like_this(DataTable arg1) throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        // For automatic transformation, change DataTable to one of
        // List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V>.
        // E,K,V must be a scalar (String, Integer, Date, enum etc)
        throw new PendingException();
    }

    @When("^player x plays in row (\\d+), column (\\d+)$")
    public void player_x_plays_in_row_column(int arg1, int arg2) throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @Then("^the board should look like this:$")
    public void the_board_should_look_like_this(DataTable arg1) throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        // For automatic transformation, change DataTable to one of
        // List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V>.
        // E,K,V must be a scalar (String, Integer, Date, enum etc)
        throw new PendingException();

    }
}
```

Turning the Table into a List of Lists

- cucumber.api.DataTable is a really rich object
- under the hood, the table is just a List of Lists of Strings: List>
- use this to create a board to support the first step


DataTable

```
Object - equals(Object) : boolean
Object - hashCode() : int
Object - toString() : String
DataTable - DataTable(List<DataTableRow>, TableConverter)
DataTable - DataTable(List<DataTableRow>, List<List<String>>, TableConverter)
DataTable - asList(Class<T>) <T> : List<T>
DataTable - asLists(Class<T>) <T> : List<List<T>>
DataTable - asMap(Class<K>, Class<V>) <K, V> : Map<K, V>
DataTable - asMaps(Class<K>, Class<V>) <K, V> : List<Map<K, V>>
DataTable - cells(int) : List<List<String>>
DataTable - diff(DataTable) : void
DataTable - diff(List<?>) : void
DataTable - diffableRows() : List<DiffableRow>
DataTable - getGherkinRows() : List<DataTableRow>
DataTable - getTableConverter() : TableConverter
DataTable - raw() : List<List<String>>
DataTable - topCells() : List<String>
DataTable - toTable(List<?>, String...) : DataTable
DataTable - transpose() : DataTable
DataTable - unorderedDiff(DataTable) : void
DataTable - unorderedDiff(List<?>) : void
```

The DataTable is a rich object

```
private List<List<String>>  board;

@Given("^a board like this:$")
public void a_board_like_this(DataTable board) throws Throwable {
    this.board=board.raw();
}

@When("^player x plays in row (\\d+), column (\\d+)$")
public void player_x_plays_in_row_column(int arg1, int arg2) thro
    System.out.println(board.toString());
    throw new PendingException();
}
```

- Use raw() to convert DataTable to List<List<String>>

Understanding DataTable

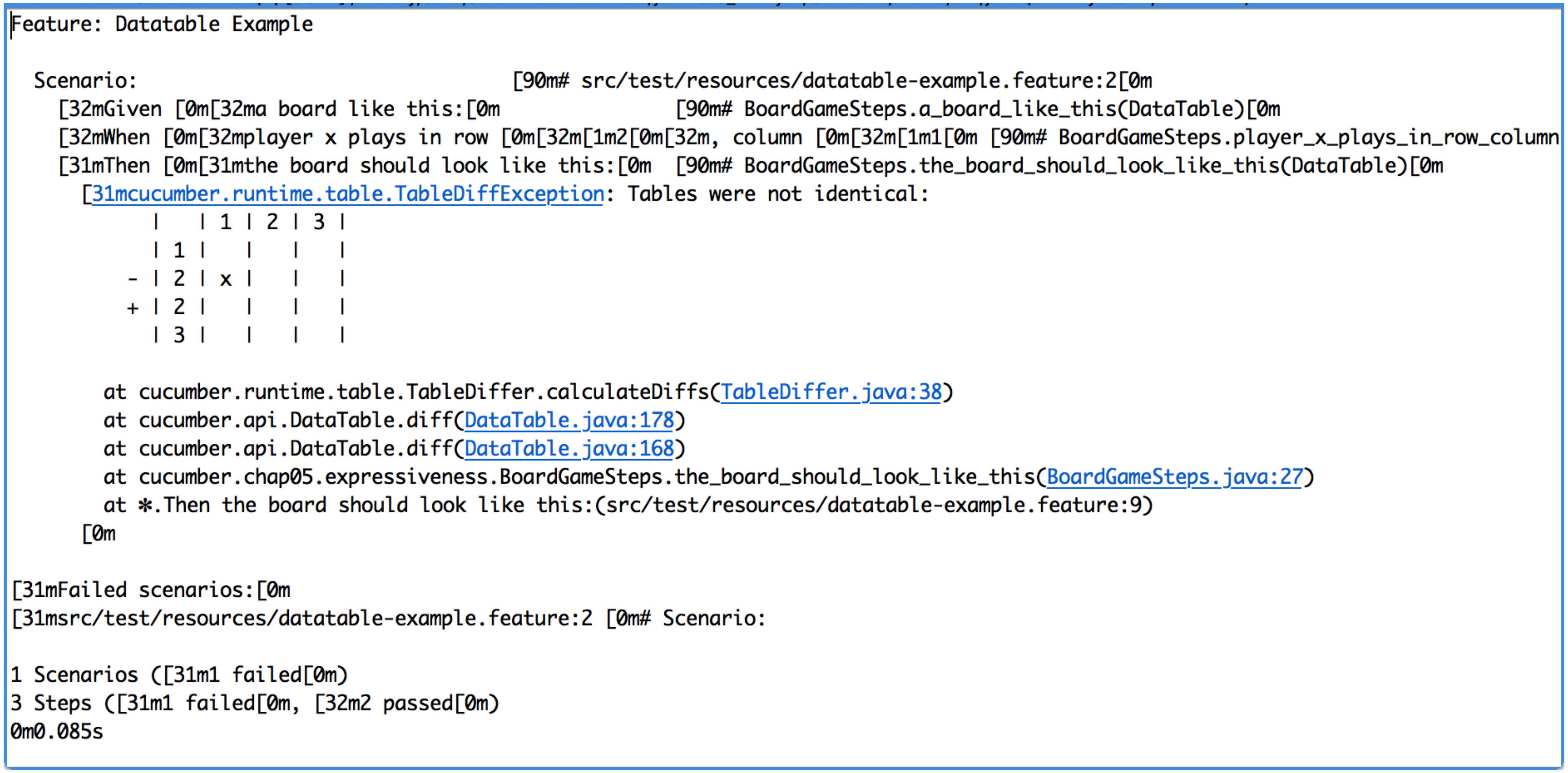
```
Feature: Datatable Example
[[, 1, 2, 3], [1, , , ], [2, , , ], [3, , , ]]
```

- Notice that the raw table includes the column and row headings

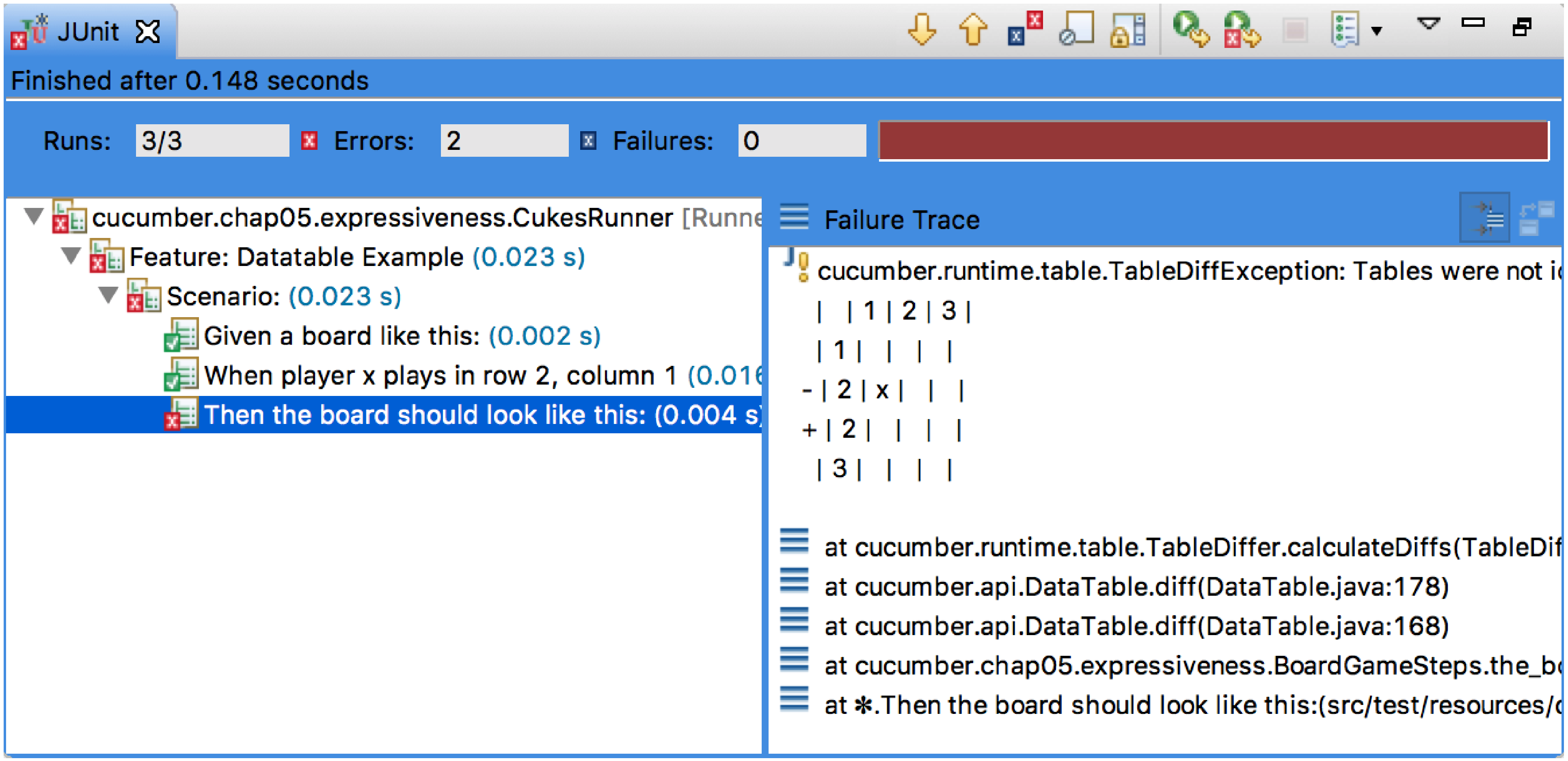
Comparing boards with diff()

- Do not implement the @When to start with a failing test

```
@When("^player x plays in row (\\d+), column (\\d+)$")
public void player_x_plays_in_row_column(int row, int column) {
   //Leave empty so that we can start with a failing test!
}

@Then("^the board should look like this:$")
public void the_board_should_look_like_this(DataTable expectedBoa
    expectedBoard.diff(board);

}
```



```
Feature: Datatable Example

  Scenario:                                    [90m# src/test/resources/datatable-example.feature:2[0m
    [32mGiven [0m[32ma board like this:[0m                [90m# BoardGameSteps.a_board_like_this(DataTable)[0m
    [32mWhen [0m[32mplayer x plays in row [0m[32m[1m2[0m[32m, column [0m[32m[1m1[0m [90m# BoardGameSteps.player_x_plays_in_row_column
    [31mThen [0m[31mthe board should look like this:[0m  [90m# BoardGameSteps.the_board_should_look_like_this(DataTable)[0m
      [31mcucumber.runtime.table.TableDiffException: Tables were not identical:
            |   | 1 | 2 | 3 |
            | 1 |   |   |   |
          - | 2 | x |   |   |
          + | 2 |   |   |   |
            | 3 |   |   |   |

        at cucumber.runtime.table.TableDiffer.calculateDiffs(TableDiffer.java:38)
        at cucumber.api.DataTable.diff(DataTable.java:178)
        at cucumber.api.DataTable.diff(DataTable.java:168)
        at cucumber.chap05.expressiveness.BoardGameSteps.the_board_should_look_like_this(BoardGameSteps.java:27)
        at ✽.Then the board should look like this:(src/test/resources/datatable-example.feature:9)
      [0m

[31mFailed scenarios:[0m
[31msrc/test/resources/datatable-example.feature:2 [0m# Scenario:

1 Scenarios ([31m1 failed[0m)
3 Steps ([31m1 failed[0m, [32m2 passed[0m)
0m0.085s
```



Finished after 0.148 seconds

Runs: 3/3 Errors: 2 Failures: 0

cucumber.chap05.expressiveness.CukesRunner [Runne
Feature: Datatable Example (0.023 s)
Scenario: (0.023 s)
Given a board like this: (0.002 s)
When player x plays in row 2, column 1 (0.01
Then the board should look like this: (0.004 s)

Failure Trace

```
cucumber.runtime.table.TableDiffException: Tables were not i
  |   | 1 | 2 | 3 |
  | 1 |   |   |   |
- | 2 | x |   |   |
+ | 2 |   |   |   |
  | 3 |   |   |   |

at cucumber.runtime.table.TableDiffer.calculateDiffs(TableDif
at cucumber.api.DataTable.diff(DataTable.java:178)
at cucumber.api.DataTable.diff(DataTable.java:168)
at cucumber.chap05.expressiveness.BoardGameSteps.the_bo
at ✻.Then the board should look like this:(src/test/resources/c
```

DataTable is not modifiable

```
@When("^player x plays in row (\\d+), column (\\d+)$")
public void player_x_plays_in_row_column(int row, int column) {
   board.get(row).set(column, "x");
}
```

- running results in:

```
java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableList.set(Collections.jav
    at cucumber.chap05.expressiveness.BoardGameSteps.player_x_pla
    at ✻.When player x plays in row 2, column 1
```

- The error happens because the DataTable is unmodifiable -> later why!

Change code to make board modifiable

```
private List<List<String>>  board=new ArrayList<>();

@Given("^a board like this:$")
public void a_board_like_this(DataTable board) {
    for(List<String> row : board.raw() ){
        List<String> newRow= new ArrayList<>();
        for(String column: row){
            newRow.add(column);
        }
        this.board.add(newRow);
    }
}
```

- read the documentation for cucumber.api.DataTable for m
- http://cukes.info/api/cucumber/jvm/javadoc/cucumber/a

Data Tables in short

- great feature of Gherkin
  - versatile
  - express data concisely
    - -> wanted in a normal specification document
- backgrounds and data tables -> can do a lot to reduce the noise and clutter in scenarios

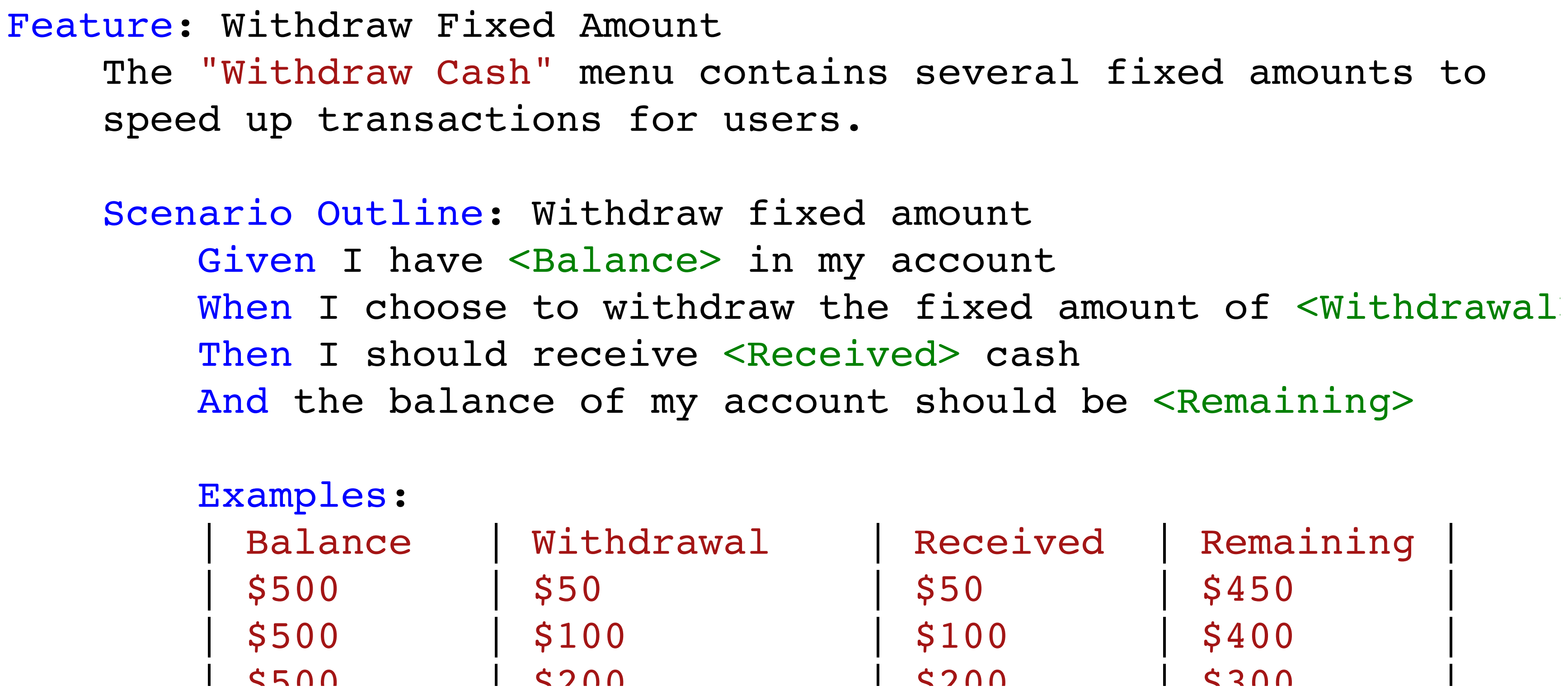
Scenario Outline

- Sometimes several scenarios follow exactly the same pattern of steps -> just with different input values or expected outcomes
  - repetition in a feature makes it boring to read
  - hard to see the essence of each scenario
- with a scenario outline: specify the steps once and then play multiple sets of values through them

An ATM scenario outline example

```
Feature: Withdraw Fixed Amount
    The "Withdraw Cash" menu contains several fixed amounts to
    speed up transactions for users.

    Scenario: Withdraw fixed amount of $50
        Given I have $500 in my account
        When I choose to withdraw the fixed amount of $50
        Then I should receive $50 cash
        And the balance of my account should be $450

    Scenario: Withdraw fixed amount of $100
        Given I have $500 in my account
        When I choose to withdraw the fixed amount of $100
        Then I should receive $100 cash
        And the balance of my account should be $400
```

Example continued



```
Feature: Withdraw Fixed Amount
    The "Withdraw Cash" menu contains several fixed amounts to
    speed up transactions for users.

    Scenario Outline: Withdraw fixed amount
        Given I have <Balance> in my account
        When I choose to withdraw the fixed amount of <Withdrawal
        Then I should receive <Received> cash
        And the balance of my account should be <Remaining>

        Examples:
        | Balance     | Withdrawal      | Received    | Remaining |
        | $500        | $50             | $50         | $450      |
        | $500        | $100            | $100        | $400      |
        | $500        | $200            | $200        | $300      |
```

- indicate placeholders within the scenario outline using angle brackets (<..>)

About scenario outlines

- a feature can have:
  - any number of Scenario Outline elements in
  - each scenario outline can have
    - any number of Examples tables
- Cucumber converts each row in the Examples table into a scenario before executing it
- Advantage scenario outline:
  - It is easy to see gaps in your examples

Bigger placeholders

- or how to make you outline more general
- substitute as much or as little as you like from any step's text

```
Scenario: Try to withdraw too much
    Given I have $100 in my account
    When I choose to withdraw the fixed amount of $200
    Then I should see an error message
    And the balance of my account should be $100
```

- compare Then step with: Then I should recieve $200
- -> can this failing scenario be part of outline?

Bigger placeholders continued,

```
Scenario Outline: Withdraw fixed amount
    Given I have <Balance> in my account
    When I choose to withdraw the fixed amount of <Withdrawal>
    Then I should <Outcome>
    And the balance of my account should be <Remaining>

    Examples:
    | Balance     | Withdrawal     | Remaining      | Outcome
    | $500        | $50            | $450           | receive $50 cas
    | $500        | $100           | $400           | receive $100 ca
    | $500        | $200           | $300           | receive $200 ca
    | $100        | $200           | $100           | see an error me
```

- use a placeholder to replace any of the text you like in a step
- the order of the placeholders in the table doesn't matter
  - -> the column header must matche the text in the placeholder

Joe asks:

How Many Examples Should I Use?

Once you have a scenario outline with a few examples, it's very easy to think of more examples, and even easier to add them. Before you know it, you have a huge, very comprehensive table of examples—and a problem.

Why?

On a system of any serious complexity, you can quite quickly start to experience what mathematicians call *combinatorial explosion*, where the number of different combinations of inputs and expected outputs becomes unmanageable. In trying to cover every possible eventuality, you end up with rows and rows of example data for Cucumber to execute. Remember that each of those little rows represents a whole scenario that might take several seconds to execute, and that can quickly start to add up. When your tests take longer to run, you slow down your feedback loop, making the whole team less productive as a result.

A really long table is also very hard to read. It's better to aim to make your examples *illustrative* or *representative* than *exhaustive*. Try to stick to what Gojko Adzic calls the *key examples.*[a] If you study the code you're testing, you'll often find that some rows of your examples table cover the same logic as another row in the table. You might also find that the test cases in your table are already covered by unit tests of the underlying code. If they're not, consider whether they should be.

Remember that readability is what's most important. If your stakeholders feel comforted by exhaustive tests, perhaps because your software operates in a safety-critical environment, then by all means put them in. Just remember that you'll never be able to prove there are no bugs. As logicians say, absence of proof is not proof of absence.

a. *Specification by Example [Ad 11]*

More than one table

```
Scenario Outline: Withdraw fixed amount
    Given I have <Balance> in my account
    When I choose to withdraw the fixed amount of <Withdrawal>
    Then I should <Outcome>
    And the balance of my account should be <Remaining>

    Examples: Successful withdrawal

    | Balance     | Withdrawal     | Remaining      | Outcome
    | $500        | $50            | $450           | receive $50 cas
    | $500        | $100           | $400           | receive $100 ca

    Examples: Attempt to withdraw too much

    | Balance     | Withdrawal     | Remaining      | Outcome
```

What is the underlying businessrule?

```
Feature: Account Creation

    Scenario Outline: Password validation
        Given I try to create an account with password "<Password
        Then I should see that the password is <Valid or Invalid>

        Examples:
        | Password    | Valid or Invalid   |
        | abc         | invalid            |
        | ab1         | invalid            |
        | abc1        | valid              |
        | abcd        | invalid            |
        | abcd1       | valid              |
```

- What is the business rule?

Make feature more self-explanatory

```
Feature: Account Creation

    Scenario Outline: Password validation
        Given I try to create an account with password "<Password
        Then I should see that the password is <Valid or Invalid>

        Examples: Too Short
        Passwords are invalid if less than 4 characters

        | Password   | Valid or Invalid   |
        | abc        | invalid            |
        | ab1        | invalid            |

        Examples: Letters and Numbers
        Passwords need both letters and numbers to be valid
```

Too much information

```
Scenario: Withdraw fixed amount of $50
    Given I have $500 in my account
    And I have pushed my card into the slot
    And I enter my PIN
    And I press "OK"
    When I choose to withdraw the fixed amount of $50
    Then I should receive $50 cash
    And the balance of my account should be $450
```

- There’s so much noise about authentication that the important part: the part about withdrawing cash is overshadowed

Finding the right level of detail

- The right level of abstraction in your scenarios -> precious skill
- different levels of detail are appropriate for different scenarios
  - -> sometimes in the same feature

```
Scenario: Successful login with PIN
    Given I have pushed my card in the slot
    When I enter my PIN
    And I press "OK"
    Then I should see the main menu
```

- Details are appropriate -> the focus is about login

Refactor

- take the three authentication steps and summarize what they do with a single high-level step:

```
Given I have authenticated with the correct PIN


@Given("^I have authenticated with the correct PIN$")
public void iHaveAuthenticatedWithTheCorrectPIN() throws Throwabl
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

//Change this into

@Given("^I have authenticated with the correct PIN$")
public void iHaveAuthenticatedWithTheCorrectPIN() throws Throwabl
    authenticateWithPIN();
}
```

Refactor

```
@Given("^I have authenticated with the correct PIN$")
public void iHaveAuthenticatedWithTheCorrectPIN() throws Throwabl
    authenticateWithPIN();
}
```

- authenticateWithPIN may make exactly the same calls as the three step definitions it is replacing but it results in a much more readable scenario:

```
Scenario: Withdraw fixed amount of $50
    Given I have $500 in my account
    And I have authenticated with the correct PIN
    When I choose to withdraw the fixed amount of $50
    Then I should receive $50 cash
    And the balance of my account should be $450
```

Doc Strings

- allows to specify a larger piece of text than fits on a single line
- example -> describe the precise content of an email message

```
Scenario: Ban Unscrupulous Users
    When I behave unscrupulously
    Then I should receive an email containing:
        """
        Dear Sir,
        Your account privileges have been revoked due to your uns
        Sincerely,
        The Management
        """
    And my account should be locked
```

Doc Strings cont.

- the entire string between the """ triple quotes is attached to the step above it
- The indentation of the opening """ is not important
- open up possibilities for specifying data in your steps i.e. specifing snippets of JSON or XML

Staying Organized with Subfolders

- start using subfolders to categorize your features
- Suggestion: use subfolders to represent different high-level tasks that a user might try to do
- Example building an intranet reporting system:

```
features/
    reading_reports/
    report_building/
    user_administration/
```

Staying Organized Subfolders

- Think about features as a book that describes what the system does
  - -> subfolders are like the chapters in that book
- So, as you tell the story of your system, what do you want the reader to see when they scan the table of contents?

Aslak says:

Features Are Not User Stories

Long ago, Cucumber started life as a tool called the RSpec Story Runner. In those days, the plain-language tests used a .story extension. When I created Cucumber, I made a deliberate decision to name the files features rather than stories. Why did I do that?

User stories are a great tool for planning. Each story contains a little bit of functionality that you can prioritize, build, test, and release. Once a story has been released, we don't want it to leave a trace in the code. We use refactoring to clean up the design so that the code absorbs the new behavior specified by the user story, leaving it looking as though that behavior had always been there.

We want the same thing to happen with our Cucumber features. The features should describe how the system behaves today, but they don't need to document the history of how it was built; that's what a version control system is for!

We've seen teams whose features directory looks like this:

```
features/
  story_38971_generate_new_report.feature
  story_38986_run_report.feature
  story_39004_log_in.feature
  ...
```

We strongly encourage you *not* to do this. You'll end up with fragmented features that just don't work as documentation for your system. One user story might map to one feature, but another user story might cause you to go and add or modify scenarios in several existing features—if the story changes the way users have to authenticate, for example. It's unlikely that there will always be a one-to-one mapping from each user story to each feature, so don't try to force it. If you need to keep a story identifier for a scenario, use a tag instead.

Staying organized with Tags

- Tags are the sticky notes you can put on pages you want to be able to find easily

```
@nightly @slow
Feature: Nightly Reports
    @widgets
    Scenario: Generate overnight widgets report

    @doofers
    Scenario: Generate overnight doofers report
```

- The scenario called Generate overnight widgets report will have three tags: @nightly, @slow, and @widgets -> the first 2 are inherited from Feature

Three main reasons for tagging scenarios:

- Documentation: You want to use a tag to attach a label to certain scenarios,for example to label them with an ID from a project management tool
- Filtering: Cucumber allows to use tags as a filter to pick out specific scenarios to run or report on
- Hooks: Run a block of code whenever a scenario with a particular tag is about to start or has just finished
- run only the scenarios tagged with @nightly:

```
$ ./cucumber --tags @nightly
```

What we just learned

- Readability should be the number-one goal when writing Gherkin features
  - sit together with a stakeholder when writing scenarios
- Use a Background to factor out repeated steps from a feature and to help tell a story
- Repetitive scenarios can be collapsed into a Scenario Outline
- Steps can be extended with multiline strings or data tables
- Organize features into subfolders, like chapters in a book