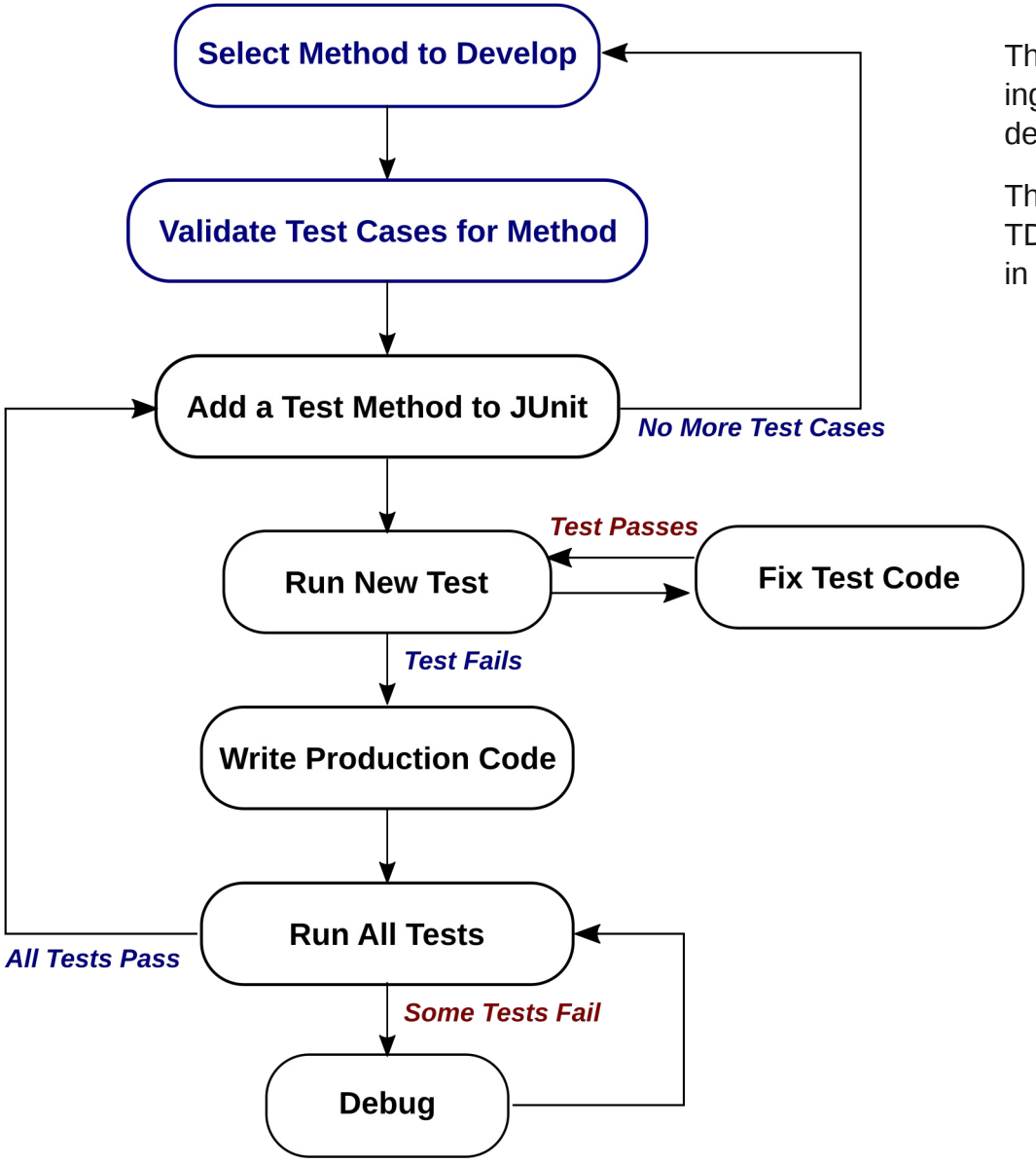
3. TDD Process Basics

Java Test Driven Development with JUnit

Module Topics

1. The TDD Process
2. The Bank Account Lab
3. Method Selection
4. Adding Test Cases
5. Writing the Code

The TDD Process



The TDD Process

This is general flow of the TDD process for writing code. There are variations of the process described by different authors.

This is also called the “red-green-refactor” TDD mantra because of the color coding used in JUnit report formatter.

Choosing a Method

- Choosing methods to implement in the right order makes development easier
- Generally develop the query methods first, then the command methods
  - *Query methods do not change the state (internal data) of an object*
  - *This means we only have to test the return value*
  - *We may decide to also test invariants to make sure nothing changed as a result of the query*
- The command methods usually change object state (internal data)
  - *Having fully tested query methods simplifies writing command JUnit tests*
  - *Fully tested query methods can be used to test the object state*

Validate Unit Tests

- Since tests guide development, bad tests results in bad code
- We do not makeup unit test cases “on the fly”
- The set of unit tests for the chosen method should already exist so that:
  - *They are consistent with the system acceptance tests*
  - *They are consistent with the design architectural constraints*
  - *The set of unit tests for the method have been validated*
- This step will be explored in more detail in the next module
  - *Just assume this step is done while for this module*

Adding a Test Case to JUnit

- Adding a test case means writing a test method in the JUnit test class to implement a chosen test case from the set of test cases

- The valid or positive test cases are all added before the invalid or negative test cases
  - *Adding the positive test cases first creates better code structure than adding positive and negative cases arbitrarily*
  - *It is more natural to have the main flow of the code handle all the logic for valid inputs*
  - *Then the invalid cases are implemented as alternate paths in the flow of the main code*
  - *Adding negative cases too early tends to produce more convoluted code structure*

Run All the Tests

- Run all of the tests and the new test should fail
  - *There is no production code to make it pass so it should fail*
  - *If it passes, we probably made an error in the test method code*
- The other possibility is the default return value from the method under development just happens to make the test pass
  - *To check this possibility, compare the expected value in the test case with the default return value of the method under development*

Write the Production Code

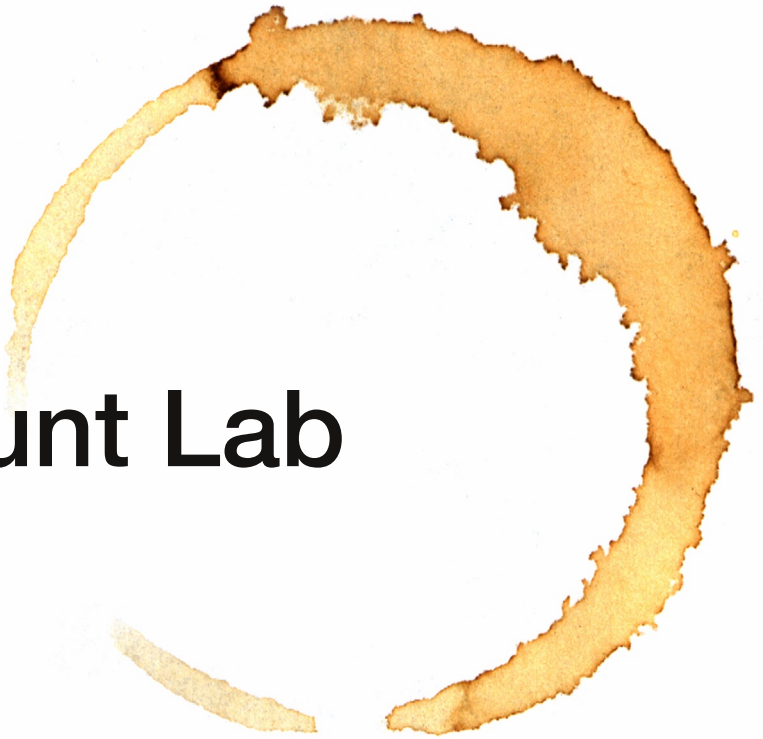
- The test case represents a class of inputs
  - *The added code should be designed to work for the class of inputs represented by the test case, not just that one test value*
  - *In the multiplication test cases below, we write code to satisfy the test classes that are represented by the test cases*
  - *The specific test cases then test to see that our logic works for that class of inputs*
  - *We should be able to use different test values from the same test classes and still have the tests pass*

Test Case	Input	Expected Output	Test Class
mult001	(2,3)	6	multiplying two positive numbers
mult002	(2,-3)	-6	multiplying a positive number and a negative number
mult003	(-2,3)	-6	multiplying a positive number and a negative number
mult004	(-2,-3)	6	multiplying two negative numbers
mult005	(-2,0)	0	multiplying a negative number by zero
mult006	(0,3)	0	multiplying a positive number by zero

Run All the Tests

- If the code is correct, all the tests should pass
  - *If the added test fails, the new code needs to be debugged*
  - *If previous test that passed no fails, adding the new code broke some of the existing code that was working previously*
- The code should be dubugged until all of the tests pass

The Bank Account Lab



The Bank Account Lab

- The bank account lab is a very simple “toy” example
  - *We will work on this problem over several modules*
  - *in this module, we will develop some query methods using TDD*
  - *In another module, we will develop the command methods using TDD*
- The full documentation for this problem is in the lab manual

The BankAccount Specification

Specification for the Bank Account class

1. The BankAccount interface provided is to be implemented as a Java class. The BankAccount object is a temporary in memory object created during an interactive session with a user. Whenever a user needs to interact with their account, a BankAccount object will be created for that purpose..

2. When instantiated, the BankAccount object will populate itself with account data from the bank mainframe database via the BankDB interface. Once the session is completed, the BankAccount object will update the database, if necessary, before the BankAccount object is destroyed.

3. Users can make deposits and withdrawals and can query their balance and available balance.

4. Each bank account has a status code associated with it which is either a 0 if the account is unencumbered or a non-zero number which indicates the account is suspended or partially suspended for some reason. No operations, excluding queries, are permitted on an account with a non-zero status.

5. Each bank account has a transaction limit and a session limit. The transaction limit is the maximum amount the user may request on a single withdrawal. The session limit is the maximum cumulative amount allowed for all withdrawals within a session (i.e. from the time the BankAccount is created until it is destroyed).

6. Users may not overdraw their accounts or exceed any limits associated with the accounts.

7. All amounts deposited or withdrawn must be greater than 0.

The BankAccount Interface

BankAccount.java

```
1  package bank;
2
3  public interface BankAccount {
4      int getBalance();
5      int getAvailBalance();
6      boolean deposit(int amt);
7      boolean withdraw(int amt);
8
9  }
10
```

The BankAccount Interface

1. The BankAccount interface has two query methods and two command methods
2. The query methods return a value but do not change any data in the account.
3. The command methods alter the internal data and return a boolean to indicate whether the command succeeded.

The BankDB Interface

BankAccount.java | BankDB.java

```
1  package bank;
2
3  public interface BankDB {
4      public int[] getData(int actnum);
5      public void putData(int actnum, int[] data);
6
7  }
8
```

The BankDB Interface

Given an account number, the getData() method returns an array of integers of length 5

data[0] contains a 0 if the account is active, or an inactive code

data[1] contains the current balance

data[2] contains the available balance

data[3] contains the per transaction limit

data[4] contains the per session limit

The putData() method takes an integer array of length 3 where

data[0] contains the account number

data[1] contains the current balance

data[2] contains the available balance

Test Data

- In order to populate our test objects, we need some test data
  - *The spreadsheet below represents some constructed test data*
  - *The process for creating test data is discussed in the next module*

Account ID	Status	Balance	Available Balance	Transaction Limit	Session Limit
1111	0	$1,000.00	$1,000.00	$100.00	$500.00
2222	1	$587.00	$346.00	$100.00	$800.00
3333	0	$897.00	$239.00	$1,000.00	$10,000.00
4444	0	$397.00	$0.00	$300.00	$1,000.00
5555	0	$0.00	$0.00	$100.00	$500.00

Method Selection

Query First

- The first two methods to be implemented are the queries
  - *The queryBalance() will be demonstrated*
  - *The queryAvailBalance() will left as an exercise*
- Examining the programming contract of the queries:
  - *There are no post-conditions because the queries do not change any data in the account object*
  - *There are no preconditions since the queries always run no matter what the status of the account is*
  - *There is an invariant: none of the internal data should change*
- Do we write tests for invariants?
  - *It depends on what we call our Good Enough Quality level*
  - *We will discuss GEQ more in the next module*

Validate Test Cases

- The validation of the data will be discussed in a later module
  - *For now, we use the test cases below assuming they have been validated*

Test Case ID	Account	Expect	Balance	Available Balance	Transaction Limit	Session Limit
BAL001	5555	$0.00	$0.00	$0.00	$100.00	$500.00
After test			$0.00	$0.00	$100.00	$500.00
BAL002	2222	$587.00	$587.00	$346.00	$100.00	$800.00
After test			$587.00	$346.00	$100.00	$800.00

BankAccount.java | BankDB.java | MyAcct.java

```
1  package bank;
2
3  public class MyAcct implements BankAccount {
4
5      private boolean status;
6      private int balance;
7      private int available_balance;
8      private int transaction_limit;
9      private int session_limit;
10     private int total_this_session;
11
12     public int getBalance() {
13         return 0;
14     }
15
16     public int getAvailBalance() {
17         return 0;
18     }
19
20     public boolean deposit(int amt) {
21         return false;
22     }
23
24     public boolean withdraw(int amt) {
25         return false;
26     }
27
28 }
29
```

Creating an Implementation Class

1. A class is defined called MyAct that implements the interface. In addition to the defined methods, the class also has instance variables to hold the data from the bank database.

2. However the problem is how we get the data into the class. We need to add a constructor that fetches the data from the database when given an account number.

BankAccount.java | BankDB.java | MyAcct.java

```
1  package bank;
2
3  public class MyAcct implements BankAccount {
4
5      public MyAcct(BankDB b, int actnum) {
6          BankDB myBank = b;
7          int[] data = null;
8          data = myBank.getData(actnum);
9          status = (data[0] == 0);
10         balance = data[1];
11         available_balance = data[2];
12         transaction_limit = data[3];
13         session_limit = data[4];
14         total_this_session = 0;
15
16     }
17
```

Implementing a Constructor

1. The constructor as implemented will now populate the account object when it is created. A dependency injection is used to provide the account with a reference to the bank database

2. The problem is that we cannot access the bank database nor should we in a development environment. What we need is an object we can use as if it were the bank database. This sort of stand-in object is called a mock object.

Mock Objects

- A mock is an object we use in a test environment that appears to be real object but is just a facade
  - *A mock object implements the same interface as the real object, but the implementation just pumps out the right replies to the test cases*
  - *Because the interface is the same as the real object, there is no way for our code to be able to tell the difference between the mock and the real object*

- Mocking is a standard technique in TDD
  - *In a later module we will look at some standard mocking libraries that simplify the use of mock objects*
  - *The next module will discuss the role and use of mocks in more detail from a testing perspective*

MockDB.java

```
1  package bank;
2
3  public class MockDB implements BankDB {
4
5          public int [] getData (int accountNumber) {
6              int[] data = null;
7              if (accountNumber == 5555) {
8                  data = new int[5];
9                  data[0] = 0;
10                 data[1] = 0;
11                 data[2] = 0;
12                 data[3] = 100;
13                 data[4] = 500;
14             }
15             else if (accountNumber == 2222) {
16                 data = new int[5];
17                 data[0] = 0;
18                 data[1] = 587;
19                 data[2] = 346;
20                 data[3] = 100;
21                 data[4] = 800;
22             }
23             return data;
24     }
25     public void putData(int actnum, int[] data) {
26         return;
27     }
28
29 }
30
```

Implementing a Mock Bank Database

1. The mock database just returns the test data for the account used in the first test case. The putData() method doesn't do anything since we do not use it in our testing.

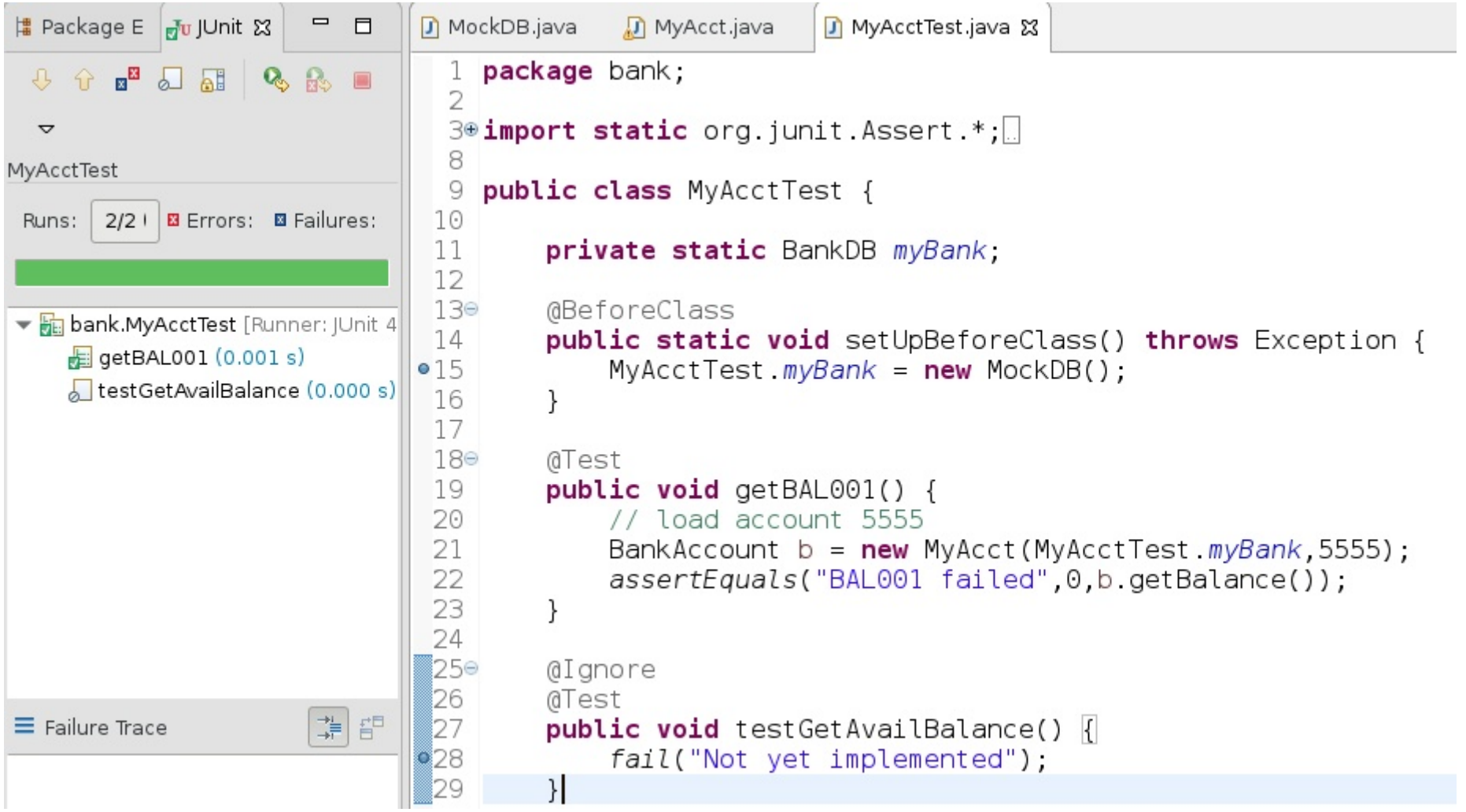
2. At this point we have added code without testing it. It is a simple matter to write a test script to see that the mock is working correctly. Whether or not we would do this as opposed to just a visual inspection is again a quality decision and is discussed in the student manual

Using the Mock Database

1. After generating the test class, we can use one of the fixture methods to create an instance of the mock bank database before any of the tests are run. Since the mock database will just get garbage collected after the test runner exits, we do not need any tear down methods.

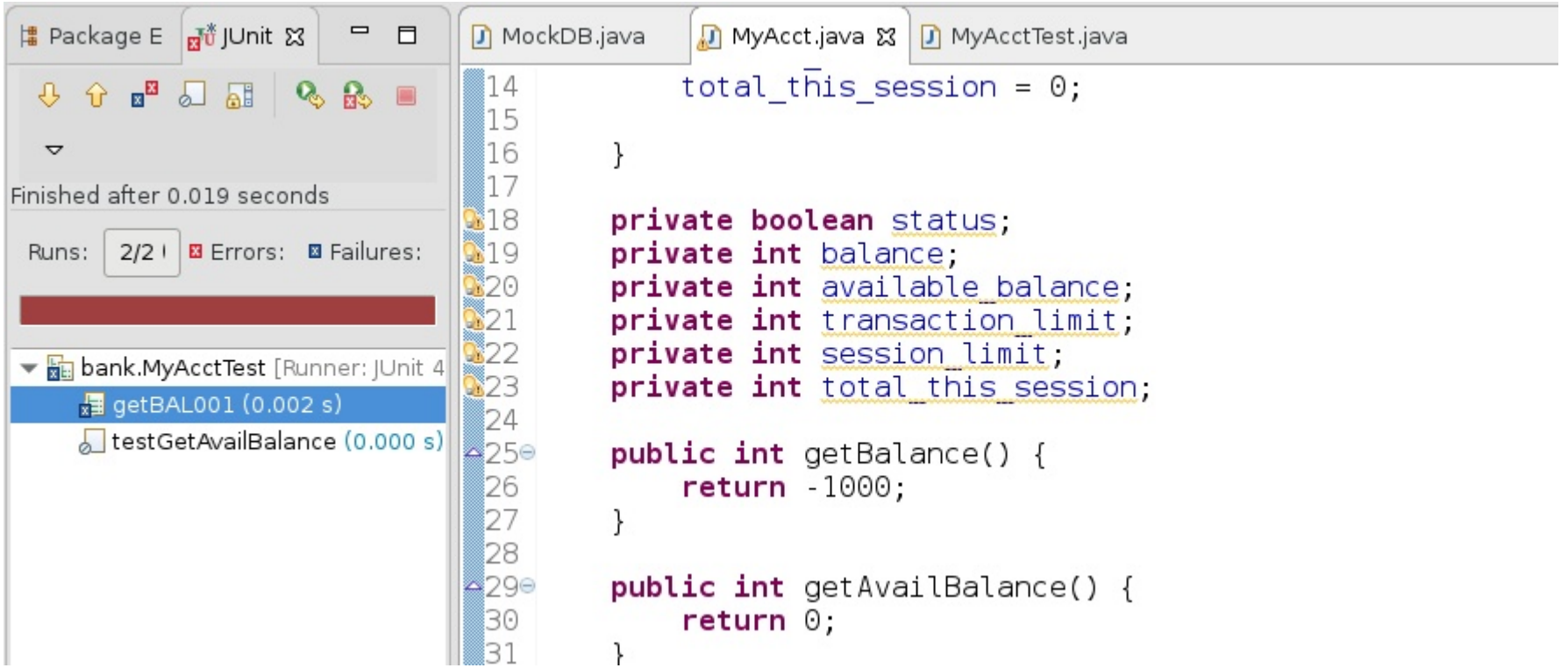
2. Because all of the tests will use the same mock object, we only need to create it once. This object is what will be passed in the dependency injection in the constructor to our account class.

Adding Test Cases



Adding Test BAL001

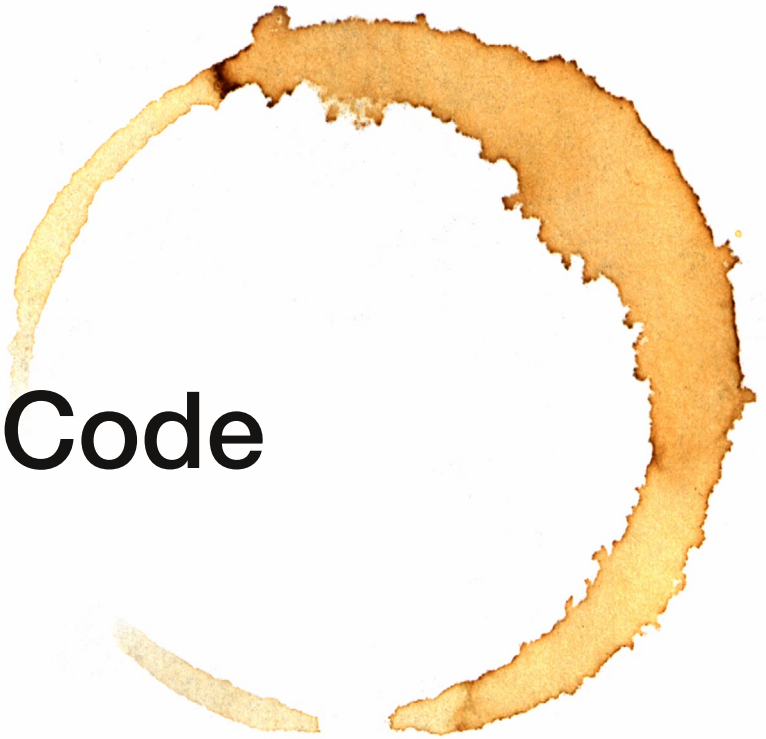
The test case is for BAL001 is added using the technique we saw for writing JUnit tests in the previous module. However when we run the test, it passes. According to our TDD protocol, it should fail.

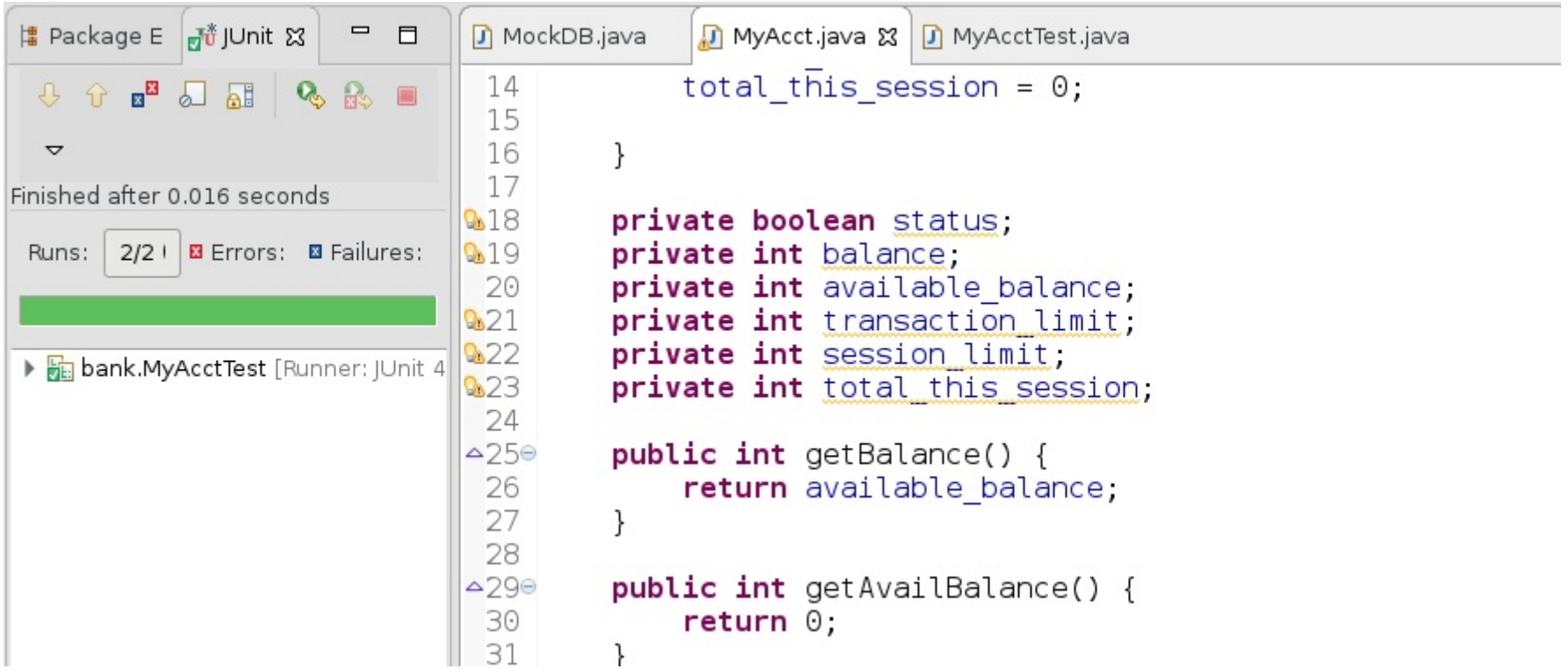


Correcting Test BAL001

The test passed because the default return value for getBalance() was 0, which as also the expected value for test. Changing the default value of the production method stub produces the required failure. While this is a contrived example, the underlying principle is to always “test the test” since there are a number of places where we could have made errors.

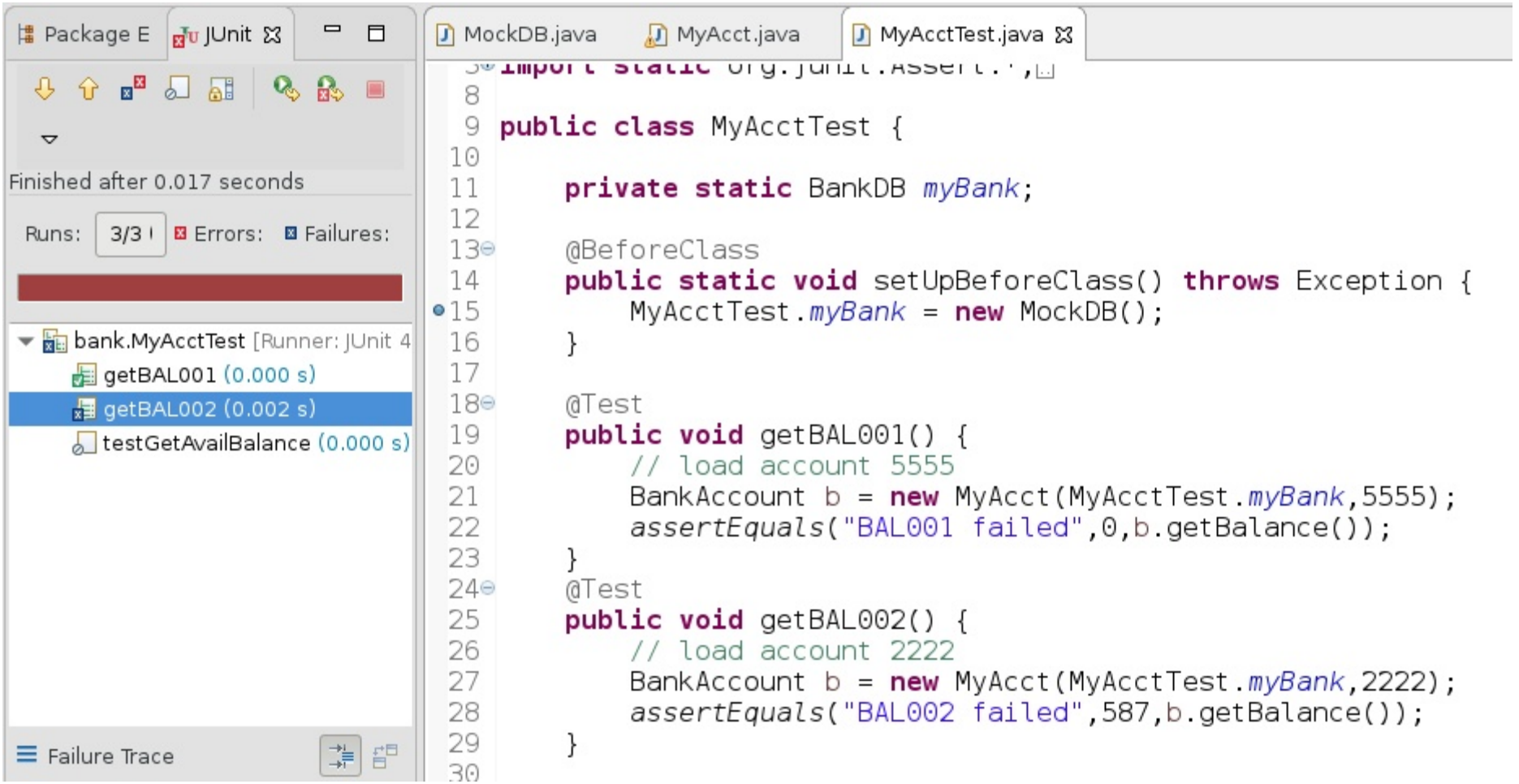
Writing the Code





Writing code

The code is written to make the test pass. However notice that the code is wrong and the test still passes. It just happens that this specific test case will not pick up the programming error. This is called coincidental correctness and an example of why do do not rely on a single test case to see if our code is correct.



Adding the New Test Case

Adding the new test case and running it picks up the error. If we had not had a robust set of test cases and just assumed that one test would be sufficient, the error would have not been caught. This is why it is important to have a robust set of unit tests to work from.

Fixing the Code

The programming error is fixed and all of the tests now pass. This example has been made simple enough that the sort of issues that we have to be aware of in the TDD process are obvious. However, in more realistic and complex coding projects, these sort of errors are not obvious on inspection. Following the TDD process has a self correcting quality that allows us to correct these errors as they occur.

End of Module 3