



*Presents*

# **Test Driven Development**

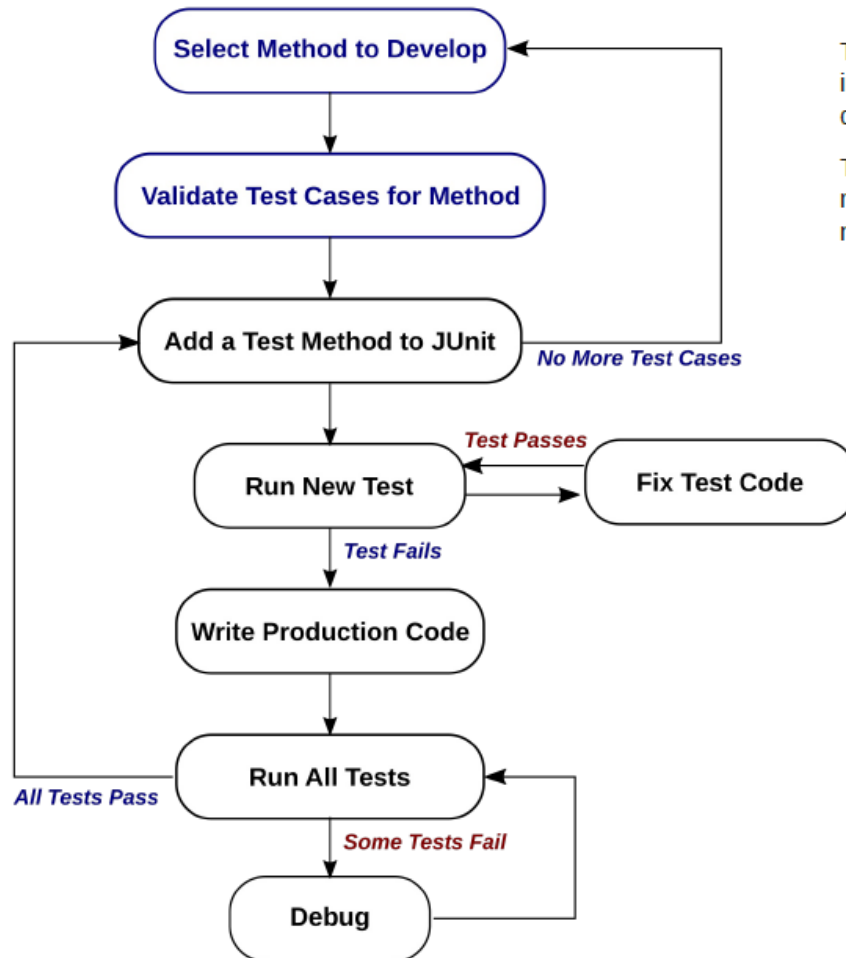
## **The TDD Process**

# 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.



# Choosing a Method

- ▶ Choosing methods to implement in the right order makes development easier
  - ✓ 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 need 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 already exist
    - 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
- ▶ For this module, we assume that all the tests have been validated

# 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 them in arbitrary order
  - ✓ 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 soon tends to produce more convoluted code

# Run All the Tests

- ▶ 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 them in arbitrary order
  - ✓ 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 soon tends to produce more convoluted code

# Write the Production Code

- ▶ The test case represents a class of inputs
  - ✓ The added code should work for the class of inputs represented by the test cases
  - ✓ 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

# Rerun All the Tests

- ▶ The production codes should pass the new test and all previous tests should pass
- ▶ If any of the tests fail, debug the code and run all the tests until they all pass





## The Bank Account Lab

# Problem Spec

## *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.

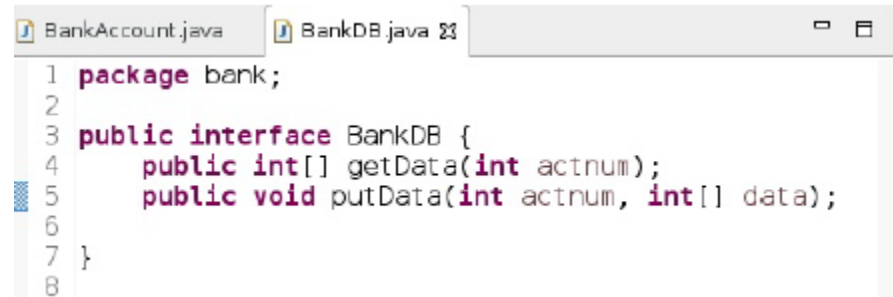
# The BankAccount Interface

```
BankAccount.java ❷
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



```
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

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

# Query First

- ▶ The first two methods to be implemented are the queries
  - ✓ The queryBalance() and the queryAvailBalance()
- ▶ Examining the programming contract of the queries:
  - ✓ There are no postconditions 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: no the internal data should be changed
- ▶ Do we write tests for invariants?
  - ✓ It depends on what we call our Good Enough Quality level

# Validate Test Cases

- ▶ 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

# Implementation Class

```
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 }
```

## *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.



# Constructor Implementation

```
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 object 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 case
  - ✓ 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 them
- ▶ Mocking is a standard technique in TDD
  - ✓ In a later module we will look at the Mockito mocking library

# Hand Crafted Bank Mock

```
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 }
```

## *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

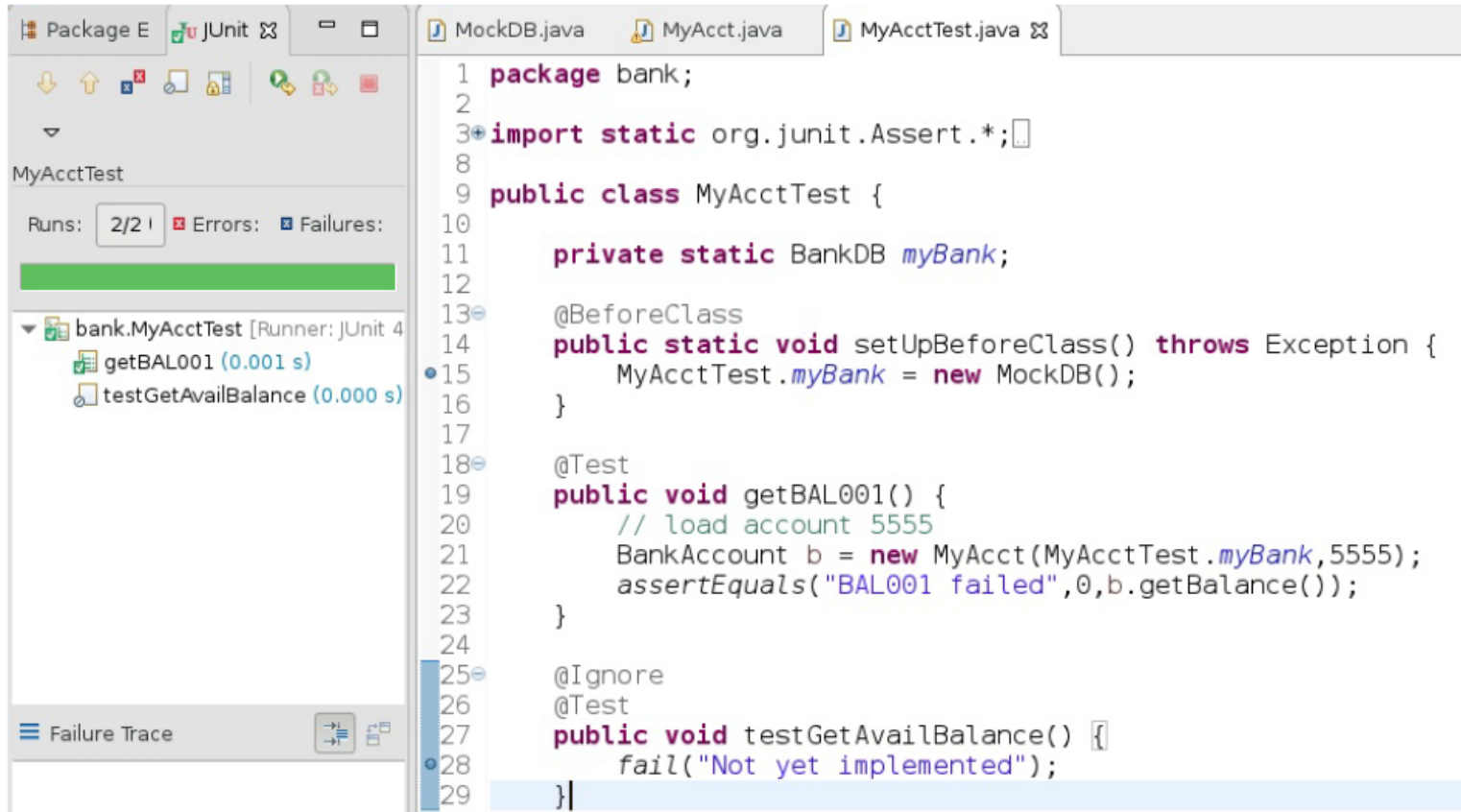
# Using the Mock

```
MockDB.java  MyAcct.java  MyAcctTest.java x
1 package bank;
2
3 import static org.junit.Assert.*;
4
5
6
7
8
9 public class MyAcctTest {
10
11     private static BankDB myBank;
12
13     @BeforeClass
14     public static void setUpBeforeClass() throws Exception {
15         MyAcctTest.myBank = new MockDB();
16     }
17 }
18
19
```

## *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 a Test Case



The screenshot shows an IDE with three tabs: MockDB.java, MyAcct.java, and MyAcctTest.java. The MyAcctTest.java tab is active, displaying the following code:

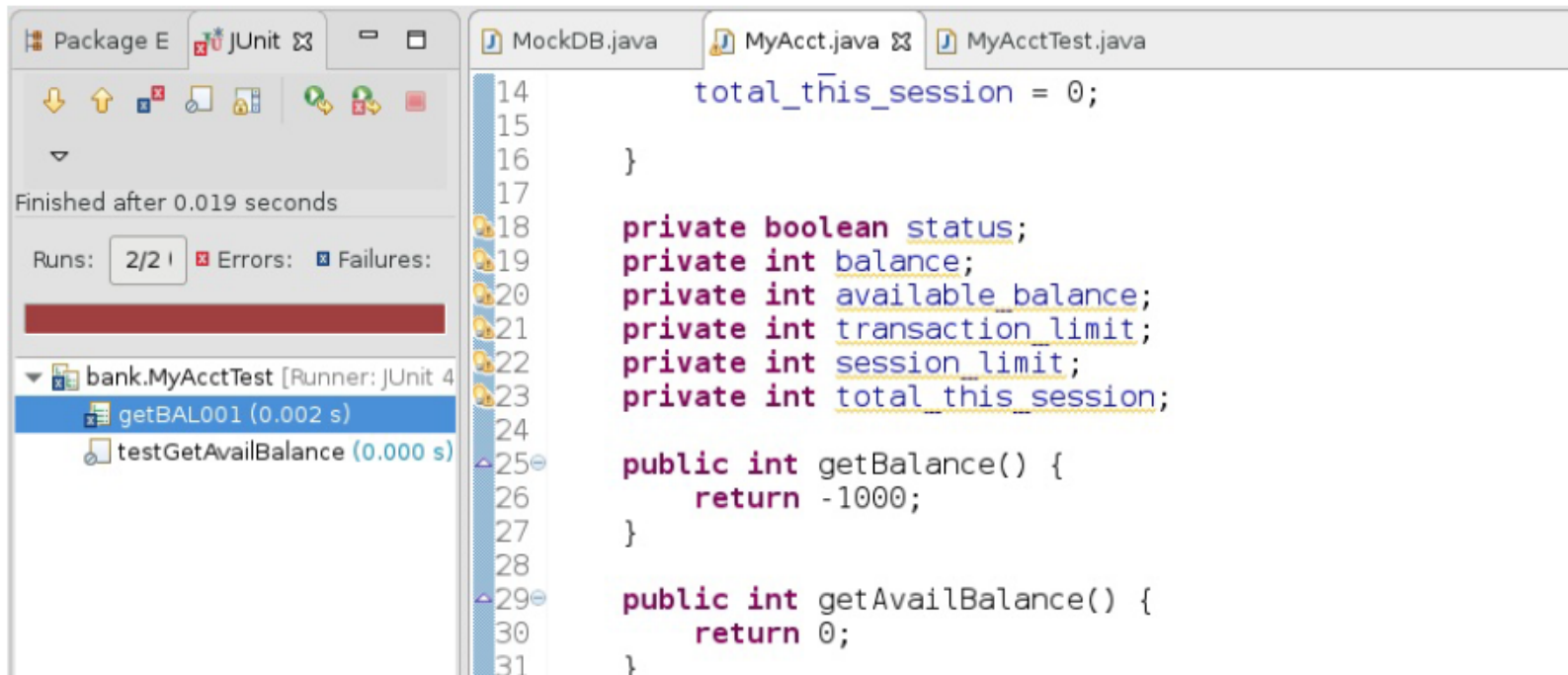
```
1 package bank;
2
3 import static org.junit.Assert.*;
4
5
6
7
8
9 public class MyAcctTest {
10
11     private static BankDB myBank;
12
13     @BeforeClass
14     public static void setUpBeforeClass() throws Exception {
15         MyAcctTest.myBank = new MockDB();
16     }
17
18     @Test
19     public void getBAL001() {
20         // load account 5555
21         BankAccount b = new MyAcct(MyAcctTest.myBank, 5555);
22         assertEquals("BAL001 failed", 0, b.getBalance());
23     }
24
25     @Ignore
26     @Test
27     public void testGetAvailBalance() {
28         fail("Not yet implemented");
29     }
30 }
```

On the left, the Package Explorer shows the package 'bank' containing the class 'MyAcctTest'. Below it, the Test Runner shows two test cases: 'getBAL001 (0.001 s)' and 'testGetAvailBalance (0.000 s)'. The 'Runs: 2/2' indicator is shown, and the 'Failure Trace' is visible at the bottom.

## Adding Test BAL001

The test case 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 the Test Case



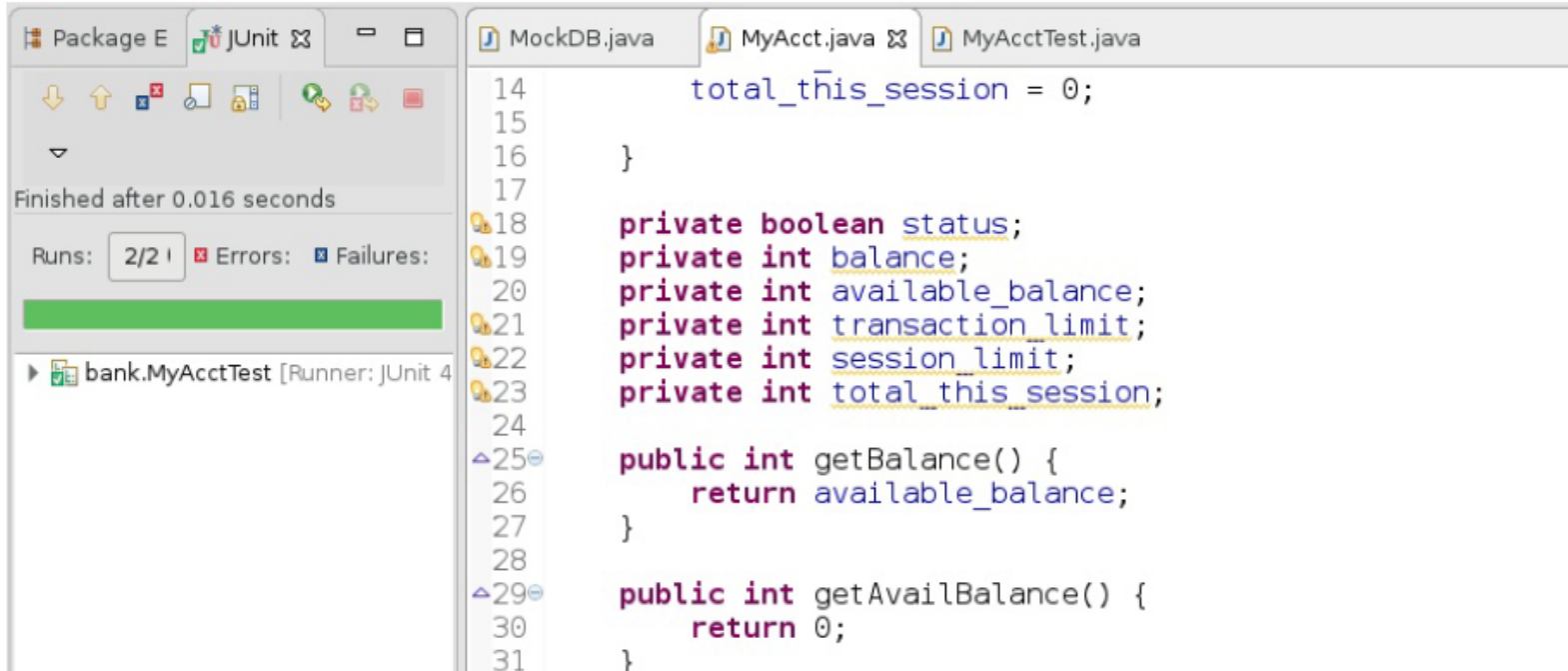
The screenshot shows an IDE with three tabs: MockDB.java, MyAcct.java, and MyAcctTest.java. On the left, a JUnit runner window shows the test results for bank.MyAcctTest. The test 'getBAL001' passed (0.002 s), but 'testGetAvailBalance' failed (0.000 s). The main editor shows the code for MyAcctTest.java, which includes a static variable 'total\_this\_session' initialized to 0, and two methods: 'getBalance()' returning -1000 and 'getAvailBalance()' returning 0.

```
14     total_this_session = 0;
15
16 }
17
18 private boolean status;
19 private int balance;
20 private int available_balance;
21 private int transaction_limit;
22 private int session_limit;
23 private int total_this_session;
24
25 public int getBalance() {
26     return -1000;
27 }
28
29 public int getAvailBalance() {
30     return 0;
31 }
```

## Correcting Test BAL001

The test passed because the default return value for `getBalance()` was 0, which is 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.

# Write Production Code



The screenshot shows an IDE with three tabs: MockDB.java, MyAcct.java, and MyAcctTest.java. On the left, a JUnit runner window shows 'Finished after 0.016 seconds' and 'Runs: 2/2 | Errors: 0 Failures: 0'. Below this, a tree view shows 'bank.MyAcctTest [Runner: JUnit 4]'. The main editor displays the code for MyAcctTest.java, which includes a static variable `total_this_session` and two methods: `getBalance()` and `getAvailBalance()`.

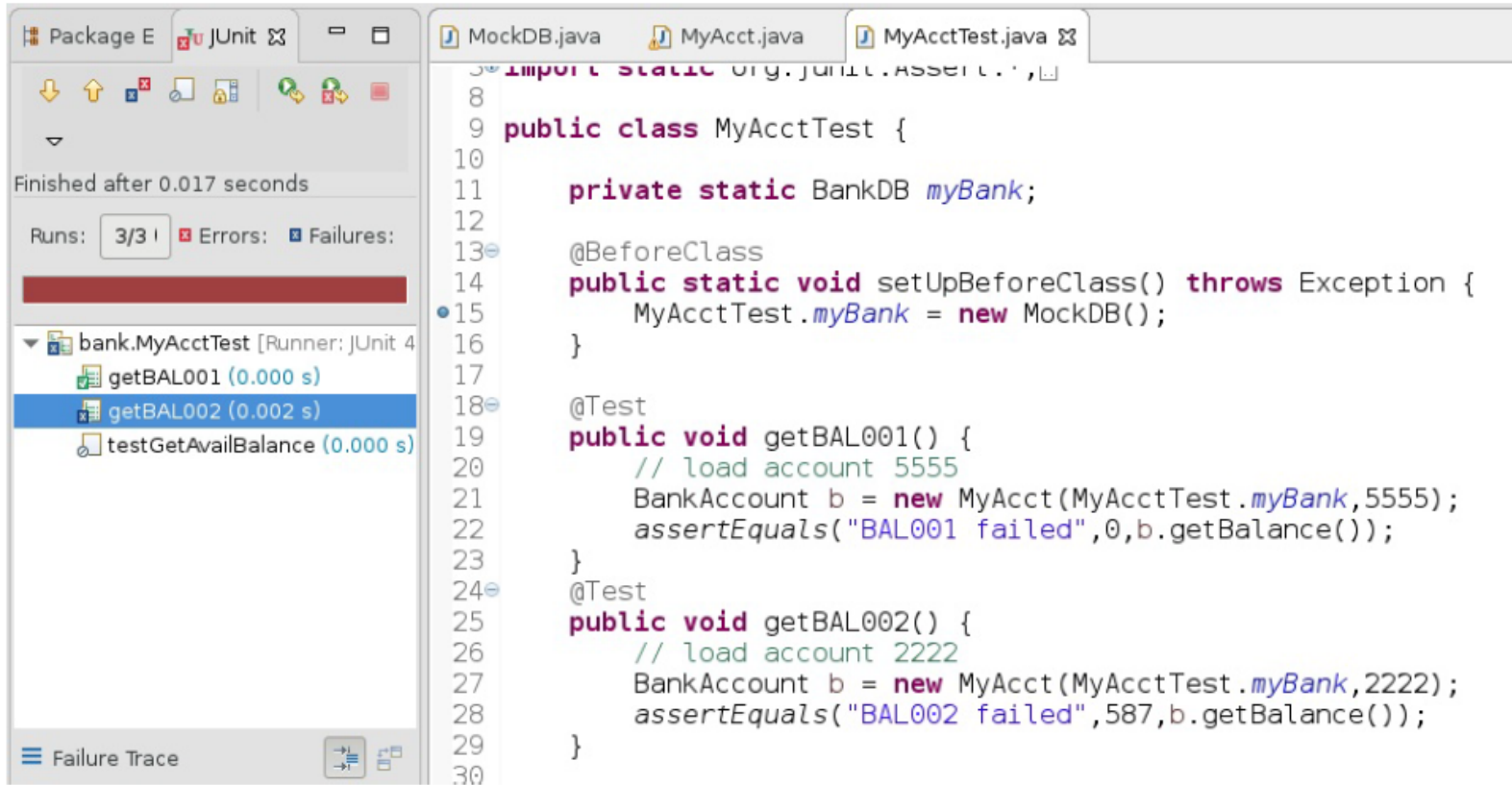
```
14         total_this_session = 0;
15
16     }
17
18     private boolean status;
19     private int balance;
20     private int available_balance;
21     private int transaction_limit;
22     private int session_limit;
23     private int total_this_session;
24
25     public int getBalance() {
26         return available_balance;
27     }
28
29     public int getAvailBalance() {
30         return 0;
31     }
```

## *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 not rely on a single test case to see if our code is correct.



# New Test = Regression Test



The screenshot displays an IDE with two main panels. The left panel shows the JUnit test runner interface, indicating that the tests finished after 0.017 seconds. It lists three tests: 'getBAL001 (0.000 s)', 'getBAL002 (0.002 s)', and 'testGetAvailBalance (0.000 s)'. The 'getBAL002' test is highlighted in blue. The right panel shows the source code for 'MyAcctTest.java'. The code includes a static field 'myBank' of type 'BankDB', a '@BeforeClass' method 'setUpBeforeClass()' that initializes 'myBank' with a new 'MockDB()' instance, and two '@Test' methods: 'getBAL001()' and 'getBAL002()'. The 'getBAL001()' method loads account 5555 and asserts that 'BAL001 failed' with a balance of 0. The 'getBAL002()' method loads account 2222 and asserts that 'BAL002 failed' with a balance of 587.

```
1  import static org.junit.Assert.*;
2
3  public class MyAcctTest {
4
5      private static BankDB myBank;
6
7      @BeforeClass
8      public static void setUpBeforeClass() throws Exception {
9          MyAcctTest.myBank = new MockDB();
10     }
11
12     @Test
13     public void getBAL001() {
14         // load account 5555
15         BankAccount b = new MyAcct(MyAcctTest.myBank, 5555);
16         assertEquals("BAL001 failed", 0, b.getBalance());
17     }
18
19     @Test
20     public void getBAL002() {
21         // load account 2222
22         BankAccount b = new MyAcct(MyAcctTest.myBank, 2222);
23         assertEquals("BAL002 failed", 587, b.getBalance());
24     }
25 }
```

## 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.



# Fix the Code



The screenshot shows an IDE with three tabs: MockDB.java, MyAcct.java, and MyAcctTest.java. On the left, a JUnit runner window shows a successful test run for bank.MyAcctTest, completed in 0.015 seconds with 3/3 runs, 0 errors, and 0 failures. The main editor displays the source code of MyAcct.java, which includes a session limit and a total for the current session.

```
14         total_this_session = 0;
15
16     }
17
18     private boolean status;
19     private int balance;
20     private int available_balance;
21     private int transaction_limit;
22     private int session_limit;
23     private int total_this_session;
24
25     public int getBalance() {
26         return balance;
27     }
28
29     public int getAvailBalance() {
30         return 0;
31     }
```

## *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.

# Implementing Commands

- ▶ Implementing Commands with TDD
  - ✓ Commands change the internal data of the object
  - ✓ Testing command return values is often trivial
- ▶ Four distinct kinds of tests:
  - ✓ Tests to ensure our code does not violate preconditions
  - ✓ Tests to ensure our code satisfies postconditions
  - ✓ Tests to ensure invariants are preserved
  - ✓ Tests to validate side effects

# The deposit() Contract

- ▶ Deposit preconditions
  - ✓ Account status must be 0
  - ✓ Deposit amount must be  $> 0$
- ▶ Deposit postconditions:
  - ✓ Balance should increase by amount
- ▶ Deposit invariants
  - ✓ Available balance remains unchanged??
  - ✓ The spec does not describe the effect of a deposit on the available balance!
  - ✓ All limits remain unchanged

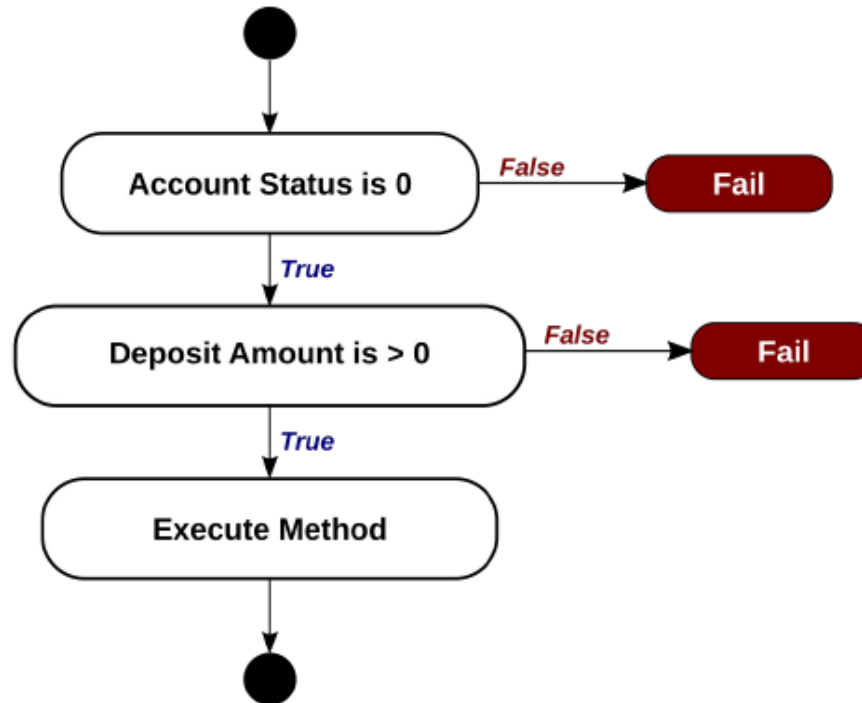
# Clarifying the Contract

- ▶ We have two alternatives
  - ✓ Deposits increase the available balance
  - ✓ Deposits do not increase the available balance
- ▶ Both alternative are reasonable but:
  - ✓ Making an assumption may result in an error
  - ✓ The only way to resolve it is to ask
- ▶ After asking the product owner we are told:
  - ✓ For fraud prevention requirements, the available balance cannot increase until the deposit is vetted overnight
  - ✓ If we had made a wrong assumption: Our tests would be wrong and our code would be wrong

# Testing Completeness for Pre-conditions

- ▶ In order to ensure we get a good set of tests:
  - ✓ We use structural testing to ensure we cover all the cases
  - ✓ We diagram the logic of all the preconditions
  - ✓ We ensure each edge is traversed by at least one test
- ▶ This is a completeness strategy:
  - ✓ It shows us the minimum number of tests needed to satisfy all the preconditions
  - ✓ For each test case postconditions and invariants still need to be checked
  - ✓ Generally, production code is written to check preconditions but rarely to validate postconditions

# The Structural Test Model



## *The Test Logic*

This is very simple since we only have two pre-conditions to worry about but it does show us that we need three tests to ensure we have considered all possible preconditions. While a model this simple could have been done on the fly, when we start to get more complex, this becomes an excellent tool for ensuring nothing has slipped through the cracks.

# The Test Cases

Test Case ID	Account	Status	Amount	Expect	Balance	Available Balance	Transaction Limit	Session Limit
DEP001	3333	0	\$1.00	TRUE	\$897.00	\$239.00	\$1,000.00	\$10,000.00
After test					\$898.00	\$239.00	\$1,000.00	\$10,000.00
DEP002	2222	1	\$1.00	FALSE	\$587.00	\$346.00	\$100.00	\$800.00
After test					\$587.00	\$346.00	\$100.00	\$800.00
DEP003	1111	0	-\$1.00	FALSE	\$1,000.00	\$1,000.00	\$100.00	\$500.00
After test					\$1,000.00	\$1,000.00	\$100.00	\$500.00

## *The Test Cases*

To fully test the pre-conditions, we can choose the three cases listed above to ensure coverage of the pre-conditions. In terms of order of implementation, we have one valid case (the one that returns true) and two invalid cases (the ones that return false) so we add the valid case first.

# Implementing A Test Case

```
9 public class MyAcctTest {
10
11     private static BankDB myBank;
12
13     @BeforeClass
14     public static void setUpBeforeClass() throws Exception {
15         MyAcctTest.myBank = new MockDB();
16     }
17
18     @Test
19     public void depositDEP001() {
20         // load account 3333
21         BankAccount b = new MyAcct(MyAcctTest.myBank, 3333);
22         // transaction accepted
23         assertTrue("DEP001 failed", b.deposit(1));
24         // post-conditions and invariants
25         assertEquals("DEP001 wrong balance", 898, b.getBalance());
26         assertEquals("DEP001 wrong avail bal", 239, b.getAvailBalance());
27     }
28 }
```

## *Implementing a Test Case*

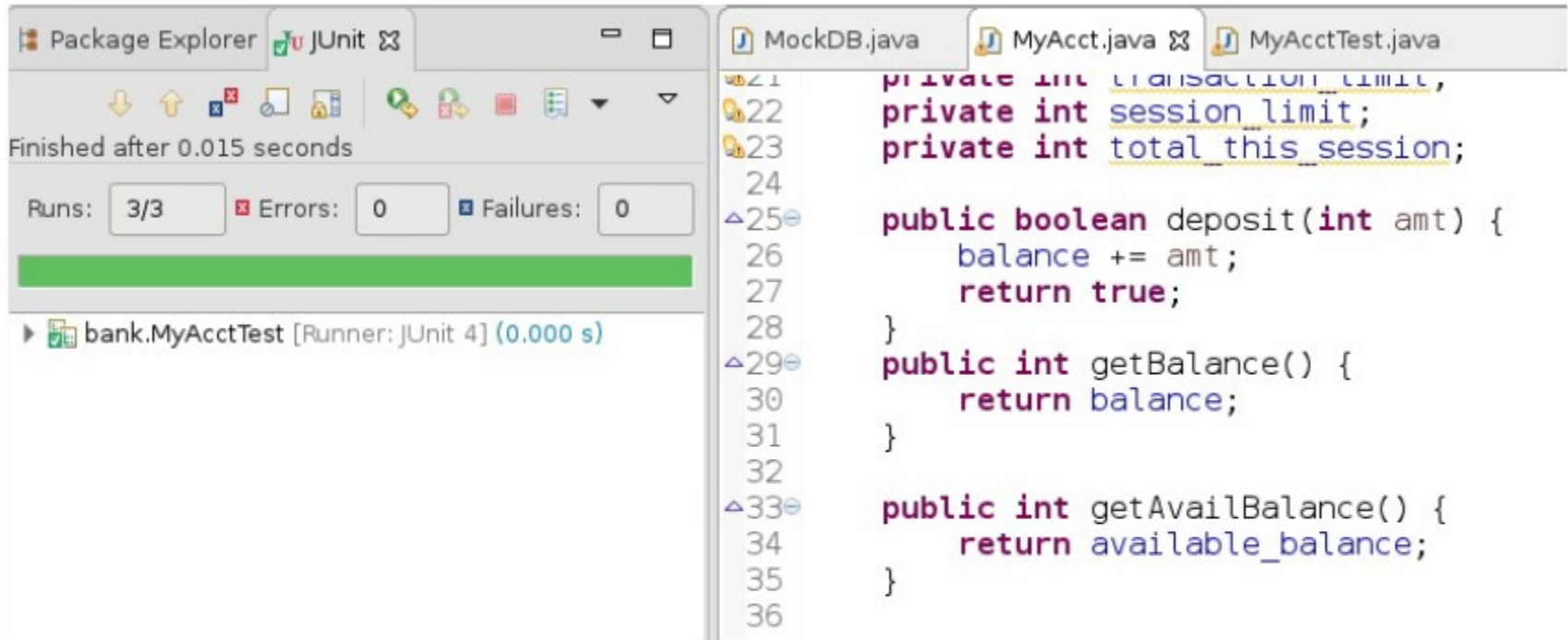
Notice the order of the assertions. We want to test first that the method executes so the first assertion checks the return value. Then we check the post-conditions and invariants that have to be true after the method executes.



# Using Multiple Assertions

- ▶ In the previous example, multiple assertions were used
  - ✓ We can place assertions anywhere in a test method we want
  - ✓ All of the assertions have to pass for the test method to pass
  - ✓ As soon as one assertion fails, the test is marked as a failure and the test method aborts

# Writing the Production Code



The screenshot shows an IDE with the Package Explorer on the left and the Java editor on the right. The Package Explorer shows a test run for `bank.MyAcctTest` [Runner: JUnit 4] (0.000 s) which finished after 0.015 seconds. The test results show 3/3 runs, 0 errors, and 0 failures. The Java editor shows the `MyAcct.java` file with the following code:

```
21 private int transaction_limit;  
22 private int session_limit;  
23 private int total_this_session;  
24  
25 public boolean deposit(int amt) {  
26     balance += amt;  
27     return true;  
28 }  
29 public int getBalance() {  
30     return balance;  
31 }  
32  
33 public int getAvailBalance() {  
34     return available_balance;  
35 }  
36
```

## Writing Production Code

After confirming that the test fails, production code is added to make the test pass.

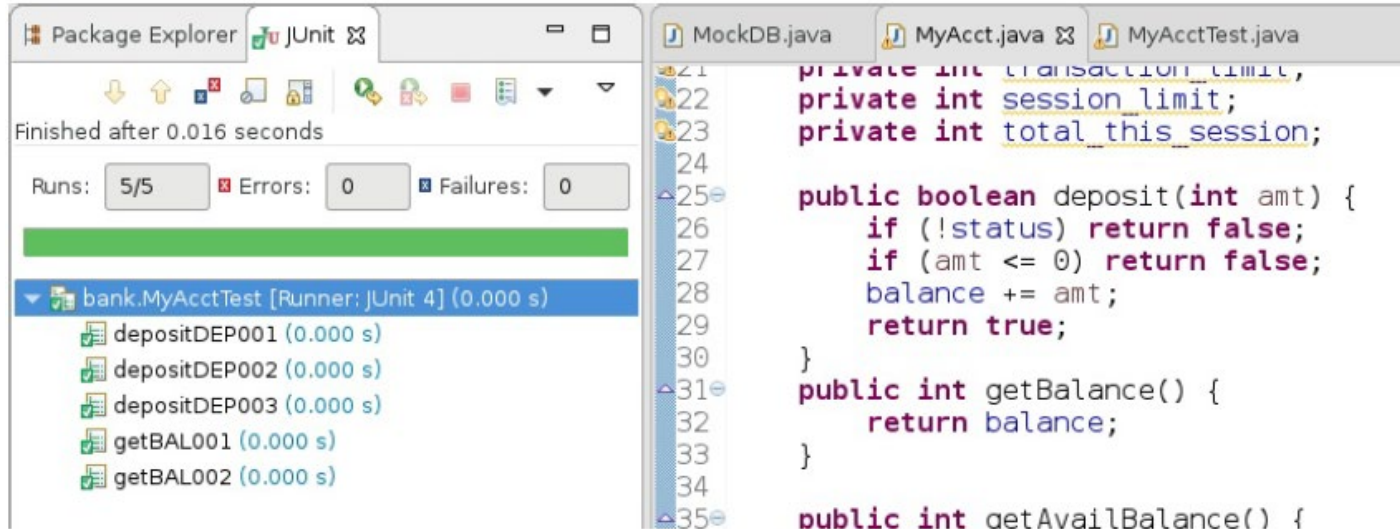
# Adding the Other Tests

```
MockDB.java  MyAcct.java  MyAcctTest.java x
26      assertEquals("DEP001 wrong avail bal",239,b.getAvailBalance());
27  }
28  @Test
29  public void depositDEP002() {
30      // load account 2222
31      BankAccount b = new MyAcct(MyAcctTest.myBank,2222);
32      // transaction accepted
33      assertFalse("DEP002 failed",b.deposit(1));
34      // post-conditions and invariants
35      assertEquals("DEP002 wrong balance",587,b.getBalance());
36      assertEquals("DEP002 wrong avail bal",346,b.getAvailBalance());
37  }
38  @Test
39  public void depositDEP003() {
40      // load account 1111
41      BankAccount b = new MyAcct(MyAcctTest.myBank,1111);
42      // transaction accepted
43      assertFalse("DEP003 failed",b.deposit(-1));
44      // post-conditions and invariants
45      assertEquals("DEP003 wrong balance",1000,b.getBalance());
46      assertEquals("DEP003 wrong avail bal",1000,b.getAvailBalance());
47  }
48  }
```

## *Writing Production Code*

After confirming that the test fails, production code is added to make the test pass.

# Running the Tests



The screenshot displays an IDE interface. On the left, the 'JUnit' tab shows test results for 'bank.MyAcctTest'. The tests are: depositDEP001 (0.000 s), depositDEP002 (0.000 s), depositDEP003 (0.000 s), getBAL001 (0.000 s), and getBAL002 (0.000 s). All tests passed. The status bar indicates 'Runs: 5/5', 'Errors: 0', and 'Failures: 0'. The main editor shows the source code for 'MyAcctTest.java'.

```
MockDB.java  MyAcct.java  MyAcctTest.java
21 private int transactionLimit;
22 private int sessionLimit;
23 private int totalThisSession;
24
25 public boolean deposit(int amt) {
26     if (!status) return false;
27     if (amt <= 0) return false;
28     balance += amt;
29     return true;
30 }
31 public int getBalance() {
32     return balance;
33 }
34
35 public int getAvailBalance() {
```

## Writing Production Code

After confirming that the test fails, production code is added to make the test pass.

# Constructors

- ▶ Constructors create objects
  - ✓ Constructors must ensure the object is in a valid initial state
  - ✓ We always test our constructors
  - ✓ But we need a specification for what a valid initial state means and how to assess it
- ▶ General OOP rule:
  - ✓ If a constructor cannot create a valid object, it throws an exception
  - ✓ For complex objects, most OOP languages will undo any results of code executed in the constructor prior to throwing the exception

# Constructor Spec for Bank Account

## *Constructor Specification*

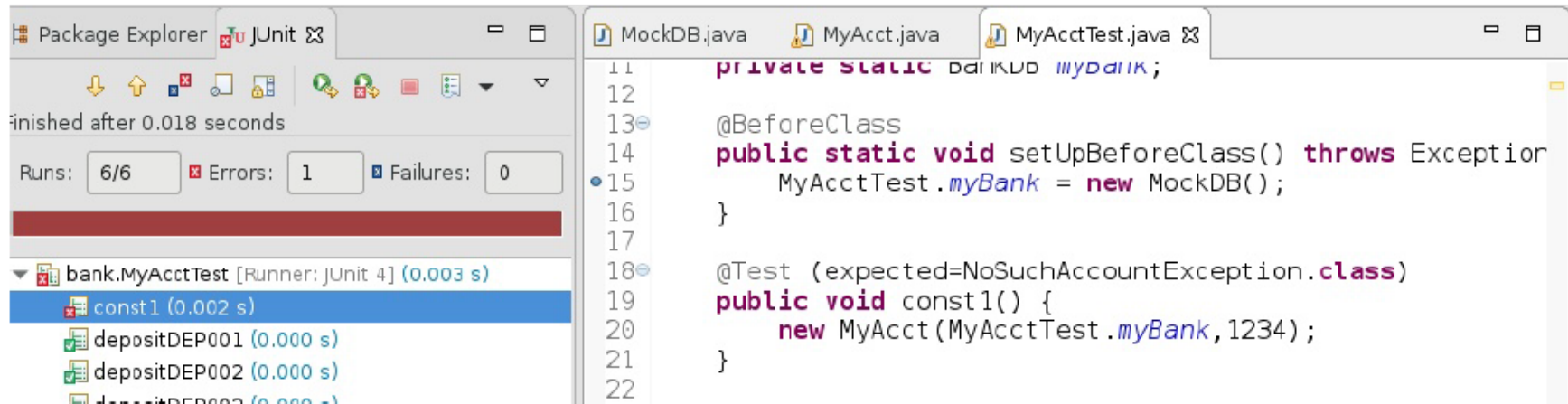
For the bank account to be valid, the following must be true:

1. The bank account must exist in the bank data base
2. All data values are  $\geq 0$
3. The available balance  $\leq$  balance
4. The transaction limit  $\leq$  session limit
5. The state is an integer from 0 to 10

Two unchecked exceptions will be used. The `NoSuchAccountException` will be thrown if the account does not exist in the bank database, and the `AccountDataException` will be thrown if any of the other rules are violated.

```
47 class NoSuchAccountException extends RuntimeException {}  
48 class AccountDataException extends RuntimeException {}  
49
```

# Adding the Constructor Test



The screenshot shows an IDE with two main panels. The left panel displays JUnit test results for `bank.MyAcctTest`. It indicates the tests finished after 0.018 seconds, with 6 runs, 1 error, and 0 failures. The test `const1` is highlighted in blue, showing a duration of 0.002 s. Below it, other tests like `depositDEP001` and `depositDEP002` are listed with green checkmarks. The right panel shows the source code for `MyAcctTest.java`. It includes a `private static` field `myBank` of type `MockDB`. The `@BeforeClass` method `setUpBeforeClass()` initializes `myBank` with a new `MockDB()` instance. The `@Test` method `const1()` is annotated with `expected=NoSuchAccountException.class` and attempts to create a `MyAcct` object using `myBank` and the value 1234.

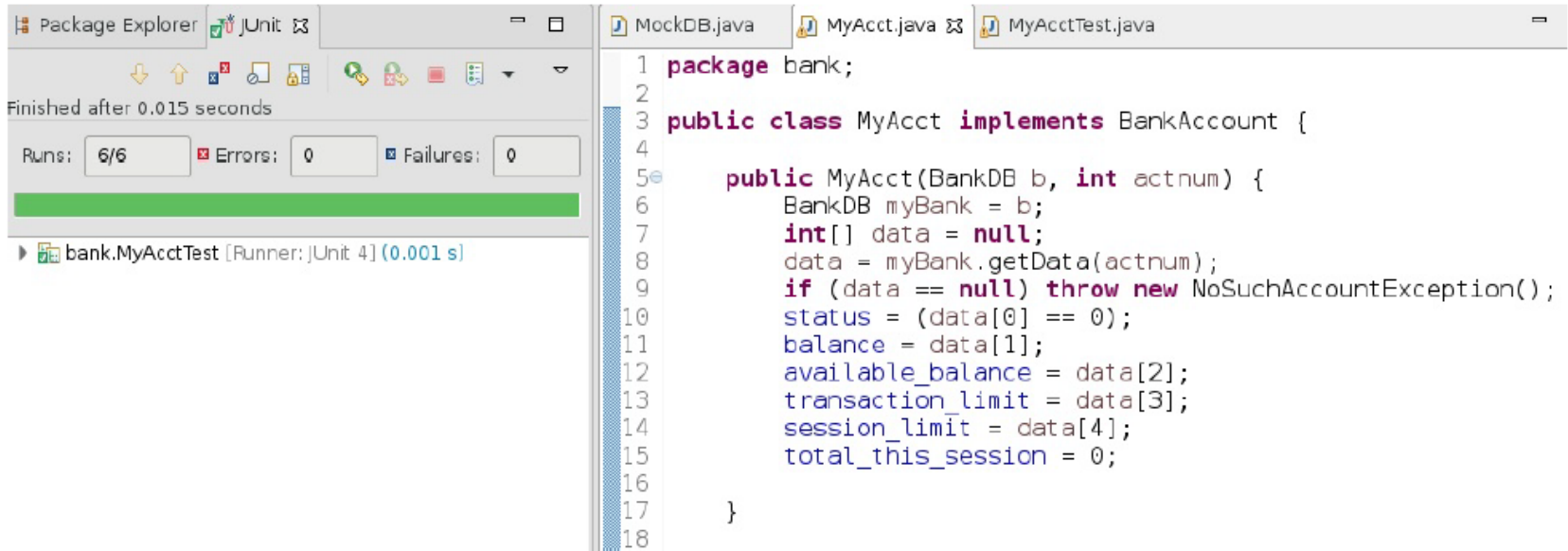
```
11 private static MockDB myBank;  
12  
13 @BeforeClass  
14 public static void setUpBeforeClass() throws Exception  
15     MyAcctTest.myBank = new MockDB();  
16 }  
17  
18 @Test (expected=NoSuchAccountException.class)  
19 public void const1() {  
20     new MyAcct(MyAcctTest.myBank, 1234);  
21 }  
22
```

## Constructor Test

The test for the `NoSuchAccountException` is added. Note that when we run it, the fact that we have a null pointer exception means that the test ends in an error rather than a failure. This often happens when we are doing exception tests because failing to throw an exception often results in some other error occurring.



# Adding the Constructor Code



The screenshot shows an IDE with three tabs: MockDB.java, MyAcct.java, and MyAcctTest.java. The MyAcct.java tab is active, showing the following code:

```
1 package bank;
2
3 public class MyAcct implements BankAccount {
4
5     public MyAcct(BankDB b, int acctnum) {
6         BankDB myBank = b;
7         int[] data = null;
8         data = myBank.getData(acctnum);
9         if (data == null) throw new NoSuchAccountException();
10        status = (data[0] == 0);
11        balance = data[1];
12        available_balance = data[2];
13        transaction_limit = data[3];
14        session_limit = data[4];
15        total_this_session = 0;
16    }
17
18 }
```

On the left, the Package Explorer shows the 'bank' package. Below it, the JUnit runner shows 'Finished after 0.015 seconds' and 'Runs: 6/6', 'Errors: 0', 'Failures: 0'. A green progress bar is visible. At the bottom left, a console window shows 'bank.MyAcctTest [Runner: JUnit 4] (0.001 s)'.

## *Adding Constructor Code*

Now we add the code in the constructor to throw the exception. If the bank account is not found, then the variable `data[]` never gets initialized and is null. Of course we would want to check to see what the real bank database returns in this case, but we have done it this way to keep the code simple.

Adding the other exception test is an exercise.



# Types of Test Cases

- ▶ Reminder: Test cases are made up of three parts
  - ✓ The test inputs
  - ✓ The expected result
  - ✓ The state the system has to be in when running the test
- ▶ Test cases are of two types:
  - ✓ Combinatorial: The test cases can be run in any order
  - ✓ Stateful: The test must be run in a particular order
- ▶ All of the tests we have been writing have been combinatorial because we are creating the object in the state we need for each test

# Changing State to Run a Test

```
MockDB.java  MyAcct.java  MyAcctTest.java  88
11  private static BankDB myBank;
12
13  @BeforeClass
14  public static void setUpBeforeClass() throws Exception {
15      MyAcctTest.myBank = new MockDB();
16  }
17
18  @Test
19  public void withdrawSession(){
20      BankAccount b = new MyAcct(MyAcctTest.myBank, 1111);
21      b.withdraw(100);
22      b.withdraw(100);
23      b.withdraw(100);
24      b.withdraw(100);
25      b.withdraw(90); // only $10 left to hit session limit
26      assertFalse(b.withdraw(20));
27      // post condition tests
28  }
29
```

## Creating a Test State

Sometimes we cannot run a test in the object's initial state. For example we cannot run a session limit test directly because the individual transaction limit is exceeded first. We need the account in the state where a single transaction will exceed the session limit.

We cannot just create an account where this is possible because of the rule that the transaction limit is always  $\leq$  the session limit. The solution is illustrated above where for account 1111, the transaction limit is \$100 and the session limit is \$500

# Questions

