5. More TDD Process

Java Test Driven Development with JUnit

Module Topics

1 Implementing Commands with TDD

2 Testing Private data

3 Testing Constructors

4 Stateful Tests

Implementing Commands with TDD

Implementing Commands

- Implementing Commands with TDD
  - *Commands change the internal data of the object*
  - *Testing command return values are often trivial*
- Four distinct kinds of tests:
  - *Tests to ensure our code does not violate pre-conditions*
  - *Tests to ensure our code satisfies post-conditions*
  - *Tests to ensure invariants are preserved*
  - *Tests to validate side effects*
- We will deal with side effect tests in a later module

The deposit() Contract

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

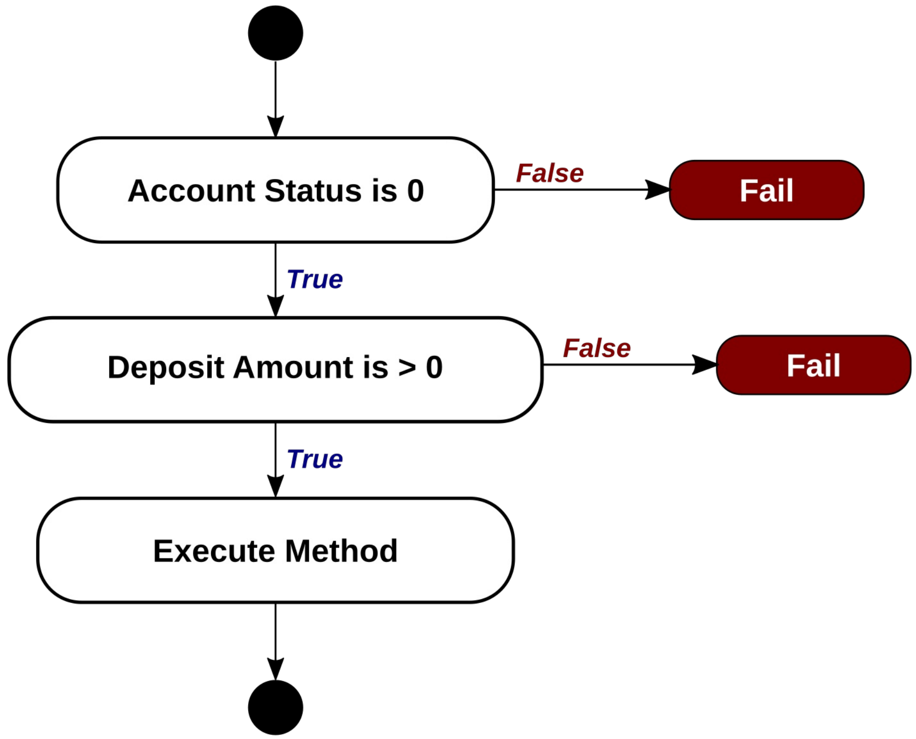
Clarifying the Contract

- 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, a deposit cannot increase the available balance, just the actual balance*
- If we had made a wrong assumption:
  - *Our tests would be wrong and our code would be wrong*

Testing Completentess 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 pre-conditions*
  - *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 pre-conditions*
- For each test case post-conditions and invariants still need to be checked
  - *Generally, production code is written to check pre-conditions but rarely to validate post-conditions*

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

```java
public class MyAcctTest {

    private static BankDB myBank;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        MyAcctTest.myBank = new MockDB();
    }

    @Test
    public void depositDEP001() {
        // load account 3333
        BankAccount b = new MyAcct(MyAcctTest.myBank,3333);
        // transaction accepted
        assertTrue("DEP001 failed",b.deposit(1));
        // post-conditions and invariants
        assertEquals("DEP001 wrong balance",898,b.getBalance());
        assertEquals("DEP001 wrong avail bal",239,b.getAvailBalance());

    }
```

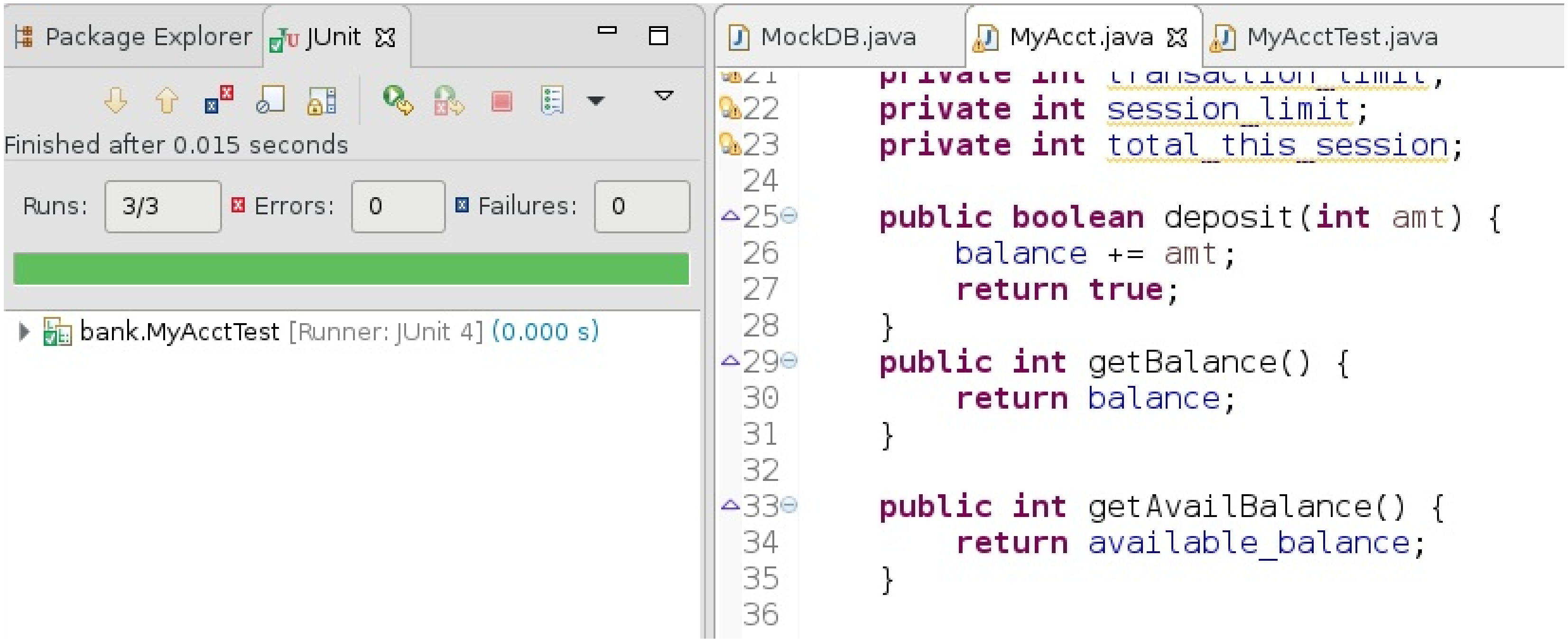
Implementing a Test Case

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



Writing Production Code

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

Adding the Other Tests

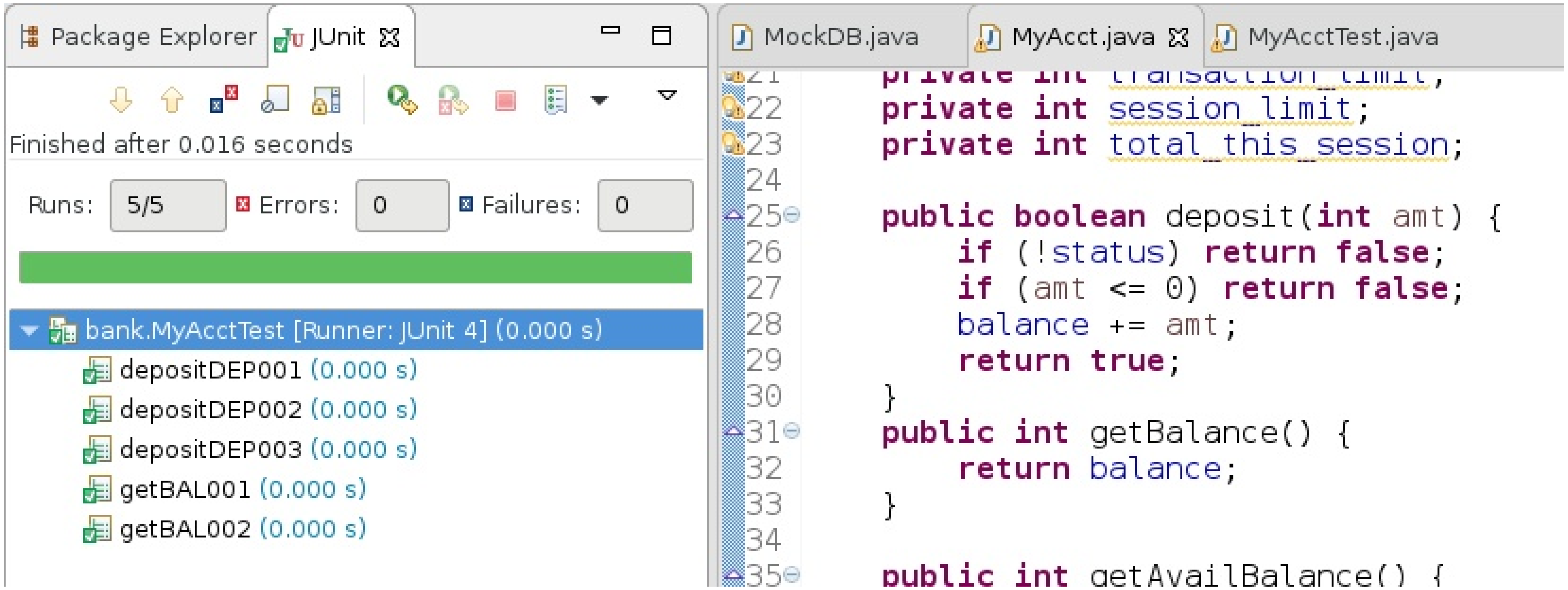
MockDB.java | MyAcct.java | MyAcctTest.java

```java
        assertEquals("DEP001 wrong avail bal",239,b.getAvailBalance());
    }
    @Test
    public void depositDEP002() {
        // load account 2222
        BankAccount b = new MyAcct(MyAcctTest.myBank,2222);
        // transaction accepted
        assertFalse("DEP002 failed",b.deposit(1));
        // post-conditions and invariants
        assertEquals("DEP002 wrong balance",587,b.getBalance());
        assertEquals("DEP002 wrong avail bal",346,b.getAvailBalance());
    }
    @Test
    public void depositDEP003() {
        // load account 1111
        BankAccount b = new MyAcct(MyAcctTest.myBank,1111);
        // transaction accepted
        assertFalse("DEP003 failed",b.deposit(-1));
        // post-conditions and invariants
        assertEquals("DEP003 wrong balance",1000,b.getBalance());
        assertEquals("DEP003 wrong avail bal",1000,b.getAvailBalance());
    }
```

Adding the Additonal Tests

The other two tests are now added and run to ensure the new tests fail. Notice the differences in the assertions from the first test.

Running the Tests



Writing Production Code

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

Testing Private Data

Attributes Versus Private Data

- The variables balance and available_balance are attributes
- Attributes represent known properties of an object
  - *Attributes have getters and/or setters*
  - *Attributes are properties described by the domain model*
  - *The data type of an attribute is define by the domain model*
- Private data refers to data that is not accessible
  - *There are no getters or setters associated with private data*
  - *The programmer decides how to represent the private data*
- Whether something is an attribute or private data is usually a design decision

Attributes Versus Private Data

- We do not test private data in TDD
  - *That function is performed by the programmer using Java language assertions in their code*
  - *This is often considered more of a debugging activity*
- There are two bad ways to test private data
  - *Make the data "temporarily" public*
  - *Add some "test only" getters for the private data*
- Both of the bad options result in a test version of the production code
  - *One of our TDD principles is there should be no test version of the production code*

Constructor Testing

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 identify 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 Specification for Bank Account

Constructor Specification

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

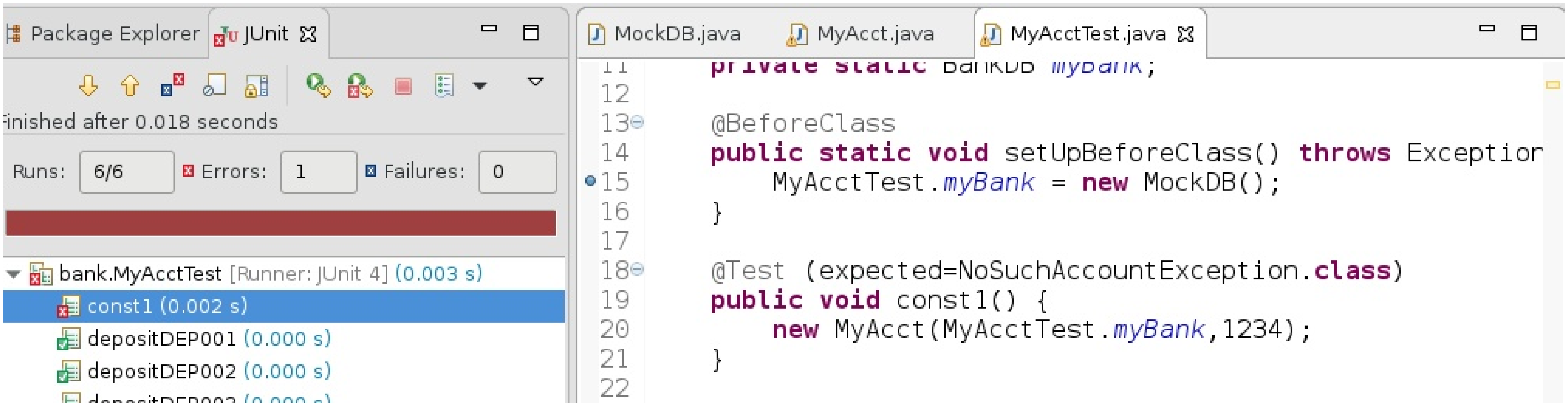
1. The bank account must exist in the bank data base
2. All data values are >= 0
3. The available balance <= balance
4. The transaction limit <= session limit
5. The state is an integer from 0 to 10

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.

```java
class NoSuchAccountException extends RuntimeException {}
class AccountDataException extends RuntimeException {}

```

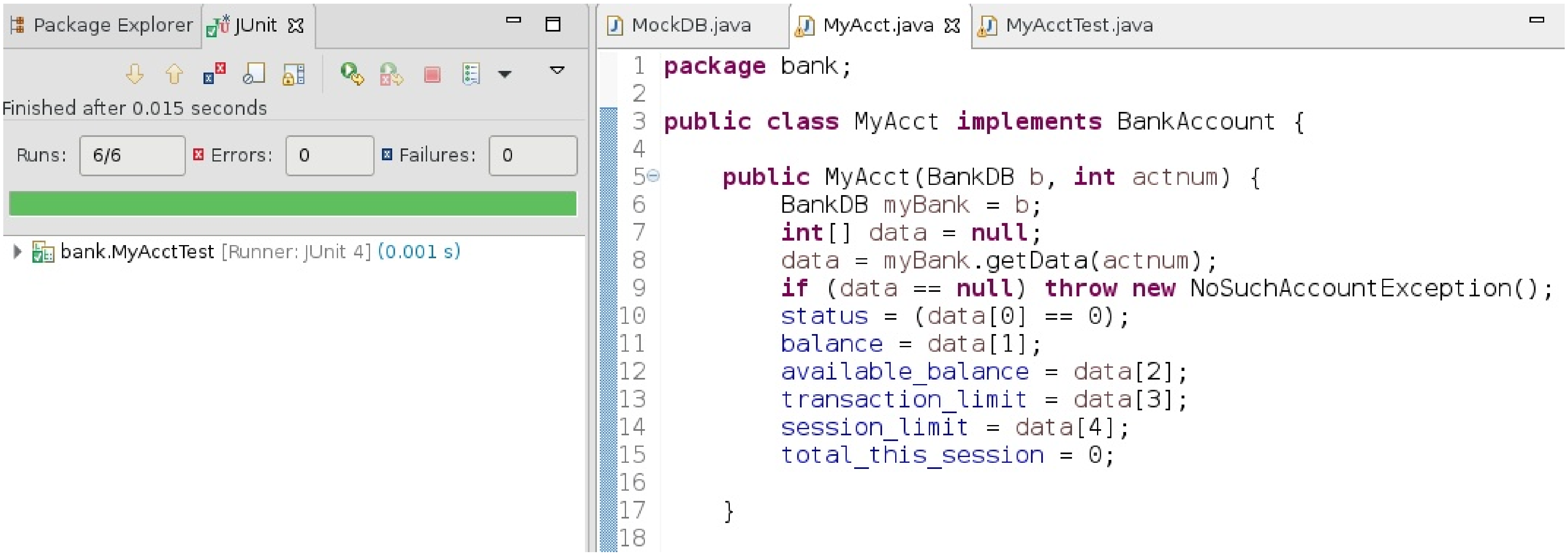
Adding the Constructor Test



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 is some other error occuring.

Adding the Constructor Code



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.

Stateful Tests

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 combinat-orial because we are creating the object in the state we need it to be in for each test

Changing State to Run a Test

MockDB.java | MyAcct.java | MyAcctTest.java

```
    private static BankDB myBank;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        MyAcctTest.myBank = new MockDB();
    }

    @Test
    public void withdrawSession(){
        BankAccount b = new MyAcct(MyAcctTest.myBank,1111);
        b.withdraw(100);
        b.withdraw(100);
        b.withdraw(100);
        b.withdraw(100);
        b.withdraw(90); // only $10 left to hit session limit
        assertFalse(b.withdraw(20));
        // post condition tests
    }
```

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 <= the session limit. The solution is illustrated above where for account 1111, the transaction limit is $100 and the session limit is $500

End of Module 3