Bank Account

Instructions

Simulate a bank account supporting opening/closing, withdrawals, and deposits of money. Watch out for concurrent transactions!

A bank account can be accessed in multiple ways. Clients can make deposits and withdrawals using the internet, mobile phones, etc. Shops can charge against the account.

Create an account that can be accessed from multiple threads/processes (terminology depends on your programming language).

It should be possible to close an account; operations against a closed account must fail.

Solution

```
class BankAccount {


    // You cannot do any operations before you open the account.

    // An account opens with a balance of 0

    // You can reopen an account

    void open() {

    }


    // you cannot do any operations after you close the account

    void close() {

    }
```

```
    // this should increment the balance

    // you cannot deposit into a closed account

    // you cannot deposit a negative amount

    void deposit(int amount) {

    }



    // this should decrement the balance

    // you cannot withdraw into a closed account

    // you cannot withdraw a negative amount

    void withdraw(int amount) {

    }



    // returns the current balance

    int getBalance() {

        return 0

    }



}
```

Instructions

Run the test file, and fix each of the errors in turn. When you get the first test to pass, go to the first pending or skipped test, and make that pass as well. When all of the tests are passing, feel free to submit.

Remember that passing code is just the first step. The goal is to work towards a solution that is as readable and expressive as you can make it.

Have fun!

Solution

```
import spock.lang.*


class BankAccountSpec extends Specification {


    def "Newly opened account has empty balance"() {

        setup:

        BankAccount account = new BankAccount()



        when:

        account.open()



        then:

        account.getBalance() == 0

    }
```

```
@Ignore
def "Can deposit money"() {
    setup:
    BankAccount account = new BankAccount()

    when:
    account.open()
    account.deposit(10)

    then:
    account.getBalance() == 10
}

@Ignore
def "Can deposit money sequentially"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
```

```
    bankAccount.open()

    bankAccount.deposit(5)

    bankAccount.deposit(23)


    then:

    bankAccount.getBalance() == 28

}


@Ignore

def "Can withdraw money"() {

    setup:

    BankAccount bankAccount = new BankAccount()


    when:

    bankAccount.open()

    bankAccount.deposit(10)

    bankAccount.withdraw(5)


    then:

    bankAccount.getBalance() == 5
```

```
}

@Ignore
def "Can withdraw money sequentially"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.deposit(23)
    bankAccount.withdraw(10)
    bankAccount.withdraw(13)

    then:
    bankAccount.getBalance() == 0
}

@Ignore
def "Cannot withdraw money from empty account"() {
    setup:
```

```
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.withdraw(5)

    then:
    thrown(Exception)
}

@Ignore
def "Cannot withdraw more money than you have"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.deposit(6)
    bankAccount.withdraw(7)

    then:
```

```
    thrown(Exception)
}

@Ignore
def "Cannot deposit negative amount"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.deposit(-1)

    then:
    thrown(Exception)
}

@Ignore
def "Cannot withdraw negative amount"() {
    setup:
    BankAccount bankAccount = new BankAccount()
```

```
    when:
    bankAccount.open()
    bankAccount.deposit(105)
    bankAccount.withdraw(-5)

    then:
    thrown(Exception)
}

@Ignore
def "Cannot get balance of closed account"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.deposit(10)
    bankAccount.close()
    bankAccount.getBalance()
```

```
    then:
    thrown(Exception)
}

@Ignore
def "Cannot deposit money into closed account"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.close()
    bankAccount.deposit(5)

    then:
    thrown(Exception)
}

@Ignore
```

```
def "Cannot withdraw money from closed account"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.deposit(20)
    bankAccount.close()
    bankAccount.withdraw(5)

    then:
    thrown(Exception)
}

@Ignore
def "Bank account is closed before it is opened"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
```

```
    bankAccount.getBalance()


    then:
    thrown(Exception)
}


@Ignore
def "Can adjust balance concurrently"() {
    setup:
    BankAccount bankAccount = new BankAccount()

    when:
    bankAccount.open()
    bankAccount.deposit(1000)
    for (int i = 0; i < 10; i++) {
        adjustBalanceConcurrently(bankAccount)
    }

    then:
    bankAccount.getBalance() == 1000
```

```
}


void adjustBalanceConcurrently(BankAccount bankAccount) {
    Random random = new Random()

    List<Thread> threads = new ArrayList<Thread>()
    (1..1000).each {
        threads.add(new Thread(
                {
                    try {
                        bankAccount.deposit(5)
                        Thread.sleep(random.nextInt(10))
                        bankAccount.withdraw(5)
                    } catch (InterruptedException ignored) {
                    } catch (Exception e) {
                        fail("Exception should not be thrown: ${e.getMessage()}")
                    }
                }
        ))
    }
```

```
        threads.each { it.start() }



        threads.each { it.join() }

    }

}
```