Method verification

Step 1

Create a BankAccount class using the following code:

```
public class BankAccount {
    private int balance;

    public BankAccountBad(int initialBalance) {
        balance = initialBalance;
    }

    public void withdraw(int amount) {
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }
}
```

This is obviously a ridiculously simplified model of a bank account, but there is enough complexity in it to show us the power of JML.

Step 2

Make our `balance` instance variable accesible to the verifier

PROF

By prepending a `/*@ spec_public @*/` expression to a variable declaration, we're telling the OpenJML verifier that the instance variable in question is part of the specification to be verified. Even though instance variables are generally private, we might want to establish facts about them, and the `/*@ spec_public @*/` expression makes them visible to the verifier.

```
public class BankAccount {
    private /*@ spec_public @*/ int balance;

    public BankAccountBad(int initialBalance) {
        balance = initialBalance;
    }

    public void withdraw(int amount) {
        balance -= amount;
    }
```

```
    public int getBalance() {
        return balance;
    }
}
```

Step 3

Make `balance` assignable in our `withdraw()` method

JML allows us to specify which variables can be mutated and which cannot. This both helps us as developers restrict our code's behaviour and helps the openJML verifier analyze that behaviour. The instance variable `balance` is mutated by `withdraw()`, so we, we can declare `balance` to be assignable by `withdraw()` using a `@ assignable VARIABLE` clause in the contract for `withdraw()`.

```
public class BankAccount {
    private /*@ spec_public @*/ int balance;

    public BankAccountBad(int initialBalance) {
        balance = initialBalance;
    }
    /*@ assignable balance; @ */
    public void withdraw(int amount) {
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }
}
```

PROF

Step 4

Add preconditions

Preconditions are the assumptions we make about the environment before we verify things. We're saying "Assuming these things are true, we can verify these other things are also true". Preconditions tell us when our contract applies. We can create preconditions using the `requires` keyword.

```
public class BankAccount {
    private /*@ spec_public @*/ int balance;

    public BankAccountBad(int initialBalance) {
        balance = initialBalance;
    }
    /*@ requires amount >= 0;
      @ assignable balance;
```

```
    @*/
    public void withdraw(int amount) {
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }
}
```

This `@ requires amount >= 0;` expression is saying "The precondition for our contract applying is that the `amount` variable is non-negative".

Step 5

Mark our getter(s) as 'pure'

A 'pure' method is one without any side effects (i.e, one which doesn't mutate any instance variables). Pure methods are very important in JML because they are the only methods in terms of which we can write our contracts. Since we want to verify claims about how the balance of our bank accounts are affected by withdrawing money, we will write our claims in terms of the behaviour of the `getBalance()` getter. To do this, we first need to explicitly mark it as pure, using the `pure` keyword:

```
public class BankAccount {
    private /*@ spec_public @*/ int balance;

    public BankAccountBad(int initialBalance) {
        balance = initialBalance;
    }
    /*@ requires amount >= 0;
      @ assignable balance;
    @*/
    public void withdraw(int amount) {
        balance -= amount;
    }
    /*@ pure @*/ public int getBalance() {
        return balance;
    }
}
```

PROF

Step 6

Add postconditions

Post conditions are conditions about the environment that we will verify are true *after* our method has run. They are the conditions we are trying to prove actually obtain. In this case, we want to verify that withdrawing money never increases the amount of money in the account.

In JML we write postconditions using `ensures` expressions of the form `//@ ensures PROPERTY`, where `PROPERTY` is some proposition we want to require be true after our method as returned.

Often, the propositions we would like to verify are true of our methods involve the previous state of some instance variables. We want to know how our method *affected* some instance variables, not just what the states of those instance variables are. To do this, we need a way of referring to the previous state of an object. That's where `\old(VARIABLE)` expressions come in. `\old(VARIABLE)` refers to the state of the instance variable `VARIABLE` before the method is called, and `VARIABLE` refers to the state of the instance variable `VARIABLE` *after* the method is called. This allows us to create contracts that specify how methods affect objects, rather than just what the state of instance variables after the method has been called.

Armed with all this new syntax, the obvious next step in our journey towards verifying our `withdraw()` method would seem writing something like the following postcondition:

```
@ ensures getBalance() <= \old(getBalance()) && getBalance() ==
\old(getBalance() - amount);
```

In english: ensure that the balance after the method is called is less than or equal to the balance the before the method is called (withdrawals never add money), and that the new balance is equal to the old balance minus the amount withdrawn.

After adding our `ensures` clause, our code looks like this:

```
public class BankAccount {
    private /*@ spec_public @*/ int balance;

    public BankAccountBad(int initialBalance) {
        balance = initialBalance;
    }
    /*@ requires amount >= 0;
      @ assignable balance;
      @ ensures getBalance() <= \old(getBalance()) && getBalance() ==
\old(getBalance() - amount)
    @*/
    public void withdraw(int amount) {
        balance -= amount;
    }
    /*@ pure @*/ public int getBalance() {
        return balance;
    }
}
```

PROF

Step 7

Perform the verification

We now have everything in place to try to verify the correctness of our method. Now, we use opeJML to do that, using the following command:

Command:

```
./openJML -esc BankAccount.java
```

Output:

```
BankAccount.java:13: verify: The prover cannot establish an assertion
(Postcondition: BankAccountBad.java:10:) in method withdraw
    public void withdraw(int amount) {
                ^
BankAccount.java:10: verify: Associated declaration:
BankAccountBad.java:13:
   @ ensures getBalance() <= \old(getBalance());
     ^
BankAccount.java:14: verify: The prover cannot establish an assertion
(ArithmeticOperationRange) in method withdraw: underflow in int
difference
        balance -= amount;
                ^
3 verification failures
```

Step 8

Read OpenJML's output after the attempted verification

Uh oh. Those errors from Step 7 don't look good! What happened?

At first, our code and contract probably seem pretty good, but there's a problem. If we stop here, then OpenJML will *not* verify that our conditions hold. Why is that? Well, because there is a loop-hole in the BankAccount class as we currently wrote it that actually makes these conditions false under certain circumstances.

PROF

The loop-hole is because we're representing the account balance with Java's int type. Since in Java ints are represented using the 2s-complement number system, they can underflow. That is, decrementing a 2s-complement number enough times eventually *increases* its value, when its value passes the minimum value that can be represented and "wraps around" to the maximum value that can be represented.

Because of this, our BankAccount class actually allows someone to *increase* their bank balance by withdrawing money! Not by withdrawing negative values of money, but by underflowing the int variable keeping track of the bank balance. JML just helped us catch a financially important logic bug! Now we just have to fix it.

Step 9

Revise the code to resolve loop-holes

There are many ways we could fix our integer underflow problem. One of the most straightfoward is to just throw an exception in the pathological case, like so:

```
public void withdraw(int amount) {
    if ((long) balance - (long) amount < Integer.MIN_VALUE) {
        throw new IllegalArgumentException("Withdrawal would cause
underflow");
    }
    balance -= amount;
}
```

In our new `withdraw()` method, we check for underflow and throw an error if we find it, and we cast our ints to longs as we check to avoid creating an underflow in our check. This takes care of our underflow issue.

Step 10

Revising our JML conditions to match our new code

We have fixed our code, but we've done so by introducing exceptions. We now need to adjust our contract to define correctness for those exceptions. Informally, our code should always throw an IllegalArgumentException when `withdraw()` is passed an underflow-producing value and never otherwise. Here's what that contract looks like in JML:

```
/*@ public normal_behavior
  @ requires amount >= 0 && ((long) balance - (long) amount) >=
Integer.MIN_VALUE;
  @ assignable balance;
  @ ensures balance == \old(balance) - amount;
  @ also
  @ public exceptional_behavior
  @ requires amount >= 0 && ((long) balance - (long) amount) <
Integer.MIN_VALUE;
  @ signals_only IllegalArgumentException;
@*/
public void withdraw(int amount) {
    if ((long) balance - (long) amount < Integer.MIN_VALUE) {
        throw new IllegalArgumentException("Withdrawal would cause
underflow");
    }
    balance -= amount;
}
```

PROF

There are a few new JML concepts here so let's unpack them. The first things to look at are these "behavior" expressions: `public normal_behavior` and `public exceptional_beahvior`. The expressions allow us to break our contract into two cases: one where an exception is thrown (i.e, `exceptional_behavior`) and one where no exceptions are thrown (i.e, `normal_behavior`). We

include both of these cases in our contract by using the `also` keyword. The `also` keyword lets us glue contracts together, requiring the both contracts be satisfied, analagous to how `&&` lets us glue predicates together, requiring that both be true for the whole expression to be true.

The last think we need to look at is the `signals_only` clause. Expressions of the form `signals_only E`, where `E` is an exception, require our code to *only* throw the exception `E`. So, our `signals_only` line is saying "the only exception allowed by this contract is IllegalArgumentException".

To summarize: we've broken our contract into two cases: one where we detect that the withdrawal amount given will produce an underflow (`exceptional_behavior`) and one where we don't (`normal_behavior`) . Each case is a contract, with its own preconditions and postconditions.

`normal_behavior`:
- Precondition: `balance` does not underflow
- Precondition: `balance` is the only variable that mutates.
- Postcondition: the new `balance` is the old `balance` minus `amount`
`exceptional_behavior`:
- Precondition: `balance` **does* underflow
- Postcondition: `IllegalArgumentException` is thrown

We join these contracts together with the `also` clause, allowing us to cover every possible case.

Final check

Having discovered a bug, fixed our code, and adjusted our contract to match our new code, we are (hopefully) in a position to prove the correctness of our code now. Run:

```
./openJML -esc BankAccount.java
```

Looking at the output, we see no errors (actually, we see no output at all). That's JML's way of telling us it succesfully verified that our code satisfies our contract. We have now formally verified our method.