Method specification

Introduction

We will begin our JML journey by specifying and then verifying a property of a method. We will imagine we work for a bank, and are interested in verifying that the software we use to model customer bank accounts is secure.

1. Create a BankAccount class

We start off by writing our naive implementation of a class for modeling bank accounts.

```
public class BankAccount {
    private int balance;

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

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

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

2. Make our `balance` instance variable accesible to the verifier

PROF

To verify the security of our software, we will want to come up with a specification that captures some notion of security. In this case, we will want to specify that the `withdraw()` method affects the `balance` instance variable in a safe way (more on that later). By default, JML does not have access to instance variables, so we must mark our `balance` variable as public (i.e, available to be used in JML specifications) using the `spec_public` keyword.

```
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;
    }
}
```

!!! note
The `@ EXPRESSION @` form is common to every JML line and distinguishes JML from java comments to **OpenJML**.

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

JML requires us to indicate which variables can be mutated by a given method and which cannot, by using the `assignable` keyword. Since `withdraw()` mutates `balance`, we must include the expression `assignable balance` in the specification of `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

4. Add preconditions

Preconditions are the assumptions we make about the environment before we verify things. We use them to verify that some property holds of some object, *under some assumption*. For example, a precondition in a method specification could be that the value of an argument variable is positive. This ability to make specifications that depend on assumptions is extremely useful, because it allows us to break up "the burden of proof" (i.e, the fact that openJML has to prove) into different components, which can make provable specifications easier to discover and easier to understand.

We can create preconditions using the `requires` keyword. We will include the precondition that the quantity of money being withdrawn from the account by `withdraw()` is non-negative. We're ignoring the danger of this unenforced assumption for the sake of simplicity.

```

```

```
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 "This specification assumes that the value of `amount` is non-negative. Under that assumption, we require that the rest of the specification is satisfied."

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:

PROF

```
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;
    }
}
```

6. Add postconditions

Postconditions are conditions about the environment that our specification requires to be true after our method has executed. When combined with precondiions, we can write specifications of the form "If this precondition obtains, then this postcondition must obtain".

We must now use some judgment to define the kind of postconditions we would like to obtain of the `withdraw()` method. One property that would be nice to have, from a security point of view, is that it's not possible to *increase* the balance of the bank account by withdrawing money from it.

In JML we write postconditions using `ensures` expressions of the form `ensures POSTCONDITION`, where `POSTCONDITION` is our postcondition .

To express our desired postcondition, we need one more piece of JML machinery. We need a way of making claims about the values of variables *before* the method is executed and the values of variables *after* the method has executed. We can do this using the `old` keyword, as in `old(VARIABLE)`. The expression `old(VARIABLE)` refers to the vavlue of the variable `VARIABLE` before the method is executed, while the expression `VARIABLE` refers to the value of that variable after the method has executed.

Armed with all this new JML, the obvious next step in our journey towards verifying our `withdraw()` method would seem writing something like the following postcondition:
`ensures getBalance() <= \old(getBalance())`

In English, this is saying "the bank balance after the amount is withdrawn cannot be bigger than before it was 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());
    @*/
    public void withdraw(int amount) {
        balance -= amount;
    }
    /*@ pure @*/ public int getBalance() {
        return balance;
    }
}
```

PROF

7. Attempt verification

At this point it might seem like we have everthing in place and the only thing left to do is feed our program into **OpenJML** for verification, but unfortunately we're not quite done yet. We *will* try to verify our program

in a moment, but we'll find that **OpenJML** isn't yet ready to accept our specification. Don't worry though, we'll fix it, but it's important to motivate the fix by considering the error messages **OpenJML** gives us.

The reason it's so important to not just jump to the correct JML code is that revising our code in response to **OpenJML** error messages is an essential part of working with JML. This tutorial wouldn't be complete if we didn't get our hands dirty with the trial-and-error revision process.

Ok, now that we're emotionally prepared for failure, let's try to verify what we've done so far by running the command `./openJML -esc BankAccount.java` .

Expected 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
```

8. Read OpenJML's output after the attempted verification

Yikes. Those errors from Step 7 look pretty bad. What happened?

PROF

At first, our code and contract probably seemed pretty reasonable, but there's a problem. There is a loop-hole in the BankAccount class as we currently wrote it that actually makes these conditions false under certain circumstances.

The loop-hole exists 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 an important logic bug! Now we just have to fix it.

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 when we're given a value of `amount` that could cause underflow.

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

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

    /*@ pure @*/ public int getBalance() {
        return balance;
    }
}
```

In our new `withdraw()` method, we check for underflow and throw an exception 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.

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.

PROF

We'll look at the JML code to do this and then unpack how it works.

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

    public BankAccountGood(int initialBalance) {
        balance = initialBalance;
    }
    /*@ 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;
    }

    /*@ pure @*/ public int getBalance() {
        return balance;
    }
}
```

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.

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.

PROF

`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.