Verifying properties of classes

In this tutorial, we will go beyond verifying properties of methods and learn how to verify properties of entire classes. We're going to prove that no matter what happens to objects in our class, our desired properties will always hold.

Taking this step will involve increasing the complexity of our contracts. We will take for granted the knowledge of method verification we gained in the first tutorial, and we will use it to prove facts about methods that we will then use to prove facts about the class as a whole.

We will expand on the banking example from the previous tutorial. This time, we will consider a bank that wants to ensure that its store of bank accounts never contains duplicate accounts.

For simplicity, we will imagine that this bank stores its list of accounts as an array of ints, each int corresponding to the account number of an account at the bank. We want to make sure this array never contains duplicates (maybe some stimulus money is supposed to be sent out to all account holders, and we don't want to deposit the stimulus twice into the same account). We will verify that it is not possible for this array to hold the same int at two different indices.

1. Write a minimal AccountRegistry class

We begin by writing the basic Java implementation of an account registry.

PROF

```
public class AccountRegistry {
    private int[] accountIDs;
    private int size;

    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

This class allows us to add and remove account IDs while maintaining the count in `size`. At this point, it behaves correctly (at least with respect to the properties we care about in this tutorial), but it does not include any formal guarantees about uniqueness.

---

2. Make instance variables visible to the verifier

We want to write specifications about `accountIDs` and `size`, but they're private. To reference them in JML, we use the `spec_public` annotation.

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
```

PROF

```
            return false;
        }
    }
```

This makes the instance variables visible in JML contracts and invariants.

---

3. Declare class invariants

We now come to the most complicated step in this tutorial.

We want to express two claims about our class:
(1) that the value of the instance variable `size` is valid (i.e, it’s never negative and never greater than the array length)
(2) that our array of bank account numbers does not contain duplicates

To do this, we need to introduce two new notions: **quantifiers** and **class invariants**.

Class invariants

Class invariants are, as the same suggests, properties of all objects belong to a class that are *invariant*. That is, they never change, no what else about those objects changes. To declare an invariant, we use the JML keyword `invariant`, as in `public invariant PROPERTY`, which declares thet the property `PROPERTY` is invariant.

Quantifiers

A quantifier is a way of making a claim about a collection. The idea comes from logic, but it corresponds to the everyday use of terms like "some" or "every" (as in, "some fruits are apples"). Any time we want to prove something of the form "there's something in this collection that has the property PROPERTY" or "everything in this collection has the property PROPERTY", we will need to use quantifiers.

PROF

In JML, quantifiers have the syntax of for-loops in java. Instead of looping through a collection and *performing an action* on each pass (as in java), we will loop through a collection and *assert a statement* on each pass.

Here's what our code looks like when we add our class invariants (the second of which involves quantifiers):

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));
    public AccountRegistry() {
        accountIDs = new int[100];
```

```
            size = 0;
        }

        public void addAccount(int id) {
            accountIDs[size] = id;
            size++;
        }

        public void removeAccount(int id) {
            int[] newIDs = new int[accountIDs.length];
            int newSize = 0;
            for (int i = 0; i < size; i++) {
                if (accountIDs[i] != id) {
                    newIDs[newSize++] = accountIDs[i];
                }
            }
            accountIDs = newIDs;
            size = newSize;
        }

        public boolean contains(int id) {
            for (int i = 0; i < size; i++) {
                if (accountIDs[i] == id) return true;
            }
            return false;
        }
    }
```

The first invariant asserts our first claim, and the second invariant asserts our second claim.

In English, the second invariant is saying "for any two indices i and j ranging over the array `accountID`, if i is not equal to j, then `accountID[i]` is not equal to `accountID[j]`, which is how we express our second claim in JML.

PROF

4. Write a contract for the constructor

We want to guarantee that the constructor properly initializes the object. That includes setting up a valid array and initializing `size`.

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));

    //@ assignable \everything;
    //@ ensures size == 0;
    //@ ensures accountIDs.length == 100;
    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

5. Write a contract for `addAccount`

PROF

We want our contract for `addAccount` to have the following features:

Preconditions

- `size` should be less than the length of the `accountIDs` array

-

Postconditions

- the `size` variable will be initialized to 0

- the size of the `addountIDs` array will be 100

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));

    //@ assignable \everything;
    //@ ensures size == 0;
    //@ ensures accountIDs.length == 100;
    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    //@ requires size < accountIDs.length;
    //@ requires !contains(id);
    //@ assignable accountIDs[size], size;
    //@ ensures accountIDs[size - 1] == id;
    //@ ensures size == \old(size) + 1;
    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

PROF

---

6. Write a contract for removeAccount

We want the contract for `removeAccount`` to have to have the following features: **Preconditions** - the `size`variable is less than the length of the`accountIDs``` array

-

We want to use `contains` inside our specs, so we must mark it as `pure`. We also write a postcondition that logically describes what it returns.

```java
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));

    //@ assignable \everything;
    //@ ensures size == 0;
    //@ ensures accountIDs.length == 100;
    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    //@ requires size < accountIDs.length;
    //@ requires !contains(id);
    //@ assignable accountIDs[size], size;
    //@ ensures accountIDs[size - 1] == id;
    //@ ensures size == \old(size) + 1;
    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    //@ ensures \result <==> (\exists int k; 0 <= k && k < size;
accountIDs[k] == id);
    public /*@ pure @*/ boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

PROF

This tells OpenJML how to reason about `contains` when used in other contracts.

8. Add a helper method for array scanning

To help verify that no duplicates remain after `removeAccount`, we'll add a helper method that checks whether a value is already in a portion of an array.

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));

    //@ assignable \everything;
    //@ ensures size == 0;
    //@ ensures accountIDs.length == 100;
    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    //@ requires size < accountIDs.length;
    //@ requires !contains(id);
    //@ assignable accountIDs[size], size;
    //@ ensures accountIDs[size - 1] == id;
    //@ ensures size == \old(size) + 1;
    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    //@ ensures \result <==> (\exists int k; 0 <= k && k < size;
accountIDs[k] == id);
    public /*@ pure @*/ boolean contains(int id) {
    //@ loop_invariant 0 <= i && i <= size;
    //@ loop_invariant (\forall int j; 0 <= j && j < i; accountIDs[j] !=
id);
```

PROF

```
    //@ decreases size - i;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

---

8. Specify `removeAccount`

This is the most complex method. We ensure that:

- `id` is in the array
- It will no longer be in the result
- No duplicates are added

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));

    //@ assignable \everything;
    //@ ensures size == 0;
    //@ ensures accountIDs.length == 100;
    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    //@ requires size < accountIDs.length;
    //@ requires !contains(id);
    //@ assignable accountIDs[size], size;
    //@ ensures accountIDs[size - 1] == id;
    //@ ensures size == \old(size) + 1;
    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
```

PROF

```
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    //@ ensures \result <==> (\exists int k; 0 <= k && k < size;
accountIDs[k] == id);
    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }

    //@ ensures \result <==> (\exists int k; 0 <= k && k < size;
accountIDs[k] == id);
    public /*@ pure @*/ boolean contains(int id) {
    //@ loop_invariant 0 <= i && i <= size;
    //@ loop_invariant (\forall int j; 0 <= j && j < i; accountIDs[j] !=
id);
    //@ decreases size - i;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

9. Add loop invariants to `removeAccount`

To help the verifier understand our method, we add loop invariants that express:

PROF

- We copy only valid elements
- We avoid `id`
- We avoid duplicates

```
public class AccountRegistry {
    private /*@ spec_public @*/ int[] accountIDs;
    private /*@ spec_public @*/ int size;

    //@ public invariant 0 <= size && size <= accountIDs.length;
    //@ public invariant (forall int i; 0 <= i && i < size;
    //@     (forall int j; 0 <= j && j < size && i != j;
    //@         accountIDs[i] != accountIDs[j]));

    //@ assignable \everything;
    //@ ensures size == 0;
    //@ ensures accountIDs.length == 100;
    public AccountRegistry() {
        accountIDs = new int[100];
        size = 0;
    }

    public void addAccount(int id) {
        accountIDs[size] = id;
        size++;
    }

    //@ requires size < accountIDs.length;
    //@ requires !contains(id);
    //@ assignable accountIDs[size], size;
    //@ ensures accountIDs[size - 1] == id;
    //@ ensures size == \old(size) + 1;
    public void removeAccount(int id) {
        int[] newIDs = new int[accountIDs.length];
        int newSize = 0;
        //@ loop_invariant 0 <= i && i <= size;
        //@ loop_invariant 0 <= newSize && newSize <= i;
        //@ loop_invariant newSize <= newIDs.length;
        //@ loop_invariant (\forall int j; 0 <= j && j < newSize;
newIDs[j] != id);
        //@ loop_invariant (\forall int a; 0 <= a && a < newSize;
        //@     (\forall int b; 0 <= b && b < newSize && a != b;
        //@         newIDs[a] != newIDs[b]));
        //@ decreases size - i;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] != id) {
                newIDs[newSize++] = accountIDs[i];
            }
        }
        accountIDs = newIDs;
        size = newSize;
    }

    //@ ensures \result <==> (\exists int k; 0 <= k && k < size;
accountIDs[k] == id);
    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }

    //@ ensures \result <==> (\exists int k; 0 <= k && k < size;
accountIDs[k] == id);
    public /*@ pure @*/ boolean contains(int id) {
    //@ loop_invariant 0 <= i && i <= size;
    //@ loop_invariant (\forall int j; 0 <= j && j < i; accountIDs[j] !=
id);
    //@ decreases size - i;
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
```

PROF

```
        }
        return false;
    }
}
```

10. Final check

At this point, you should be able to run OpenJML:

```
openjml -esc AccountRegistry.java
```

If you’ve followed along and written the contracts correctly, OpenJML should verify that:

- The class invariants are preserved
- The methods meet their contracts
- No duplicate account IDs are ever stored

You’ve now written a verified class invariant in JML!