Specifying class invariants

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, those objects will always satisfy our desired properties. These properties are called "invariant properties".

We will continue the banking theme from the first example. 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 a registry of its 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 we send out apply interest to accounts using this registry, and we don't want to apply the interest more than once to the same account). We will verify two properties of this array always hold:

Unique ID invariant

It is not possible for this array to hold the same int at two different indices.

Size Bound invariant

The instance variable we will use to track the size of the array does not exceed the lenbth of the array

---

1. Write a minimal AccountRegistry class

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

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

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

    public void addAccount(int id) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
    }

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

PROF

```
    }

    public boolean isFull() {
        return size == accountIDs.length;
    }
}
```

Just like in the last task, we're sacrificing a lot of realism for simplicity in this example.

This class allows us to add IDs, check whether an ID is in our registry, and check whether our registry is full. We are, however, not able to remove IDs (that would take us into too many complications). Our class tracks the number of stored IDs using the variable `size`. At this point, this class behaves correctly (at least with respect to the properties we care about in this tutorial), but it does not yet include any formal guarantees of correctness.

---

2. Make instance variables visible to the verifier

We want to write specifications that refer to `accountIDs` and `size`, but they're private. As we learned in the first task, we must annotate these variables with the `spec_public` keyword if we want them to be available for use in our specifications.

```
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) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
    }

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

    public boolean isFull() {
        return size == accountIDs.length;
    }
```

PROF

```
}
```

---

3. Specify our class invariants

Class invariants are defined using the `invariant` keyword, in expressions of the form `public invariant PROPERTY`, where `PROPERTY` is some expression that expresses a property.

In task 1, our specifications only referred to particular values, but to express our class invariants we will need a way to refer to *collections* of values. The feature JML makes available for this puprose is the quantifier. A quantifier is an expression that refers to a collection of values, as in the phrases "Every apple is a fruit", or "Some insects are bees". In JML, we can express statements asserting a property holds over *every* thing in some category using the `\forall` keyword, and we can express statements asserting a property holds over *some* thing in some category using the keyword `\exists`. Since one of the class invariants we want to specify refers to a collection of values (i.e, "there is no duplicated account ID among the collection of account IDs in the registry"), we need a quantifier to express it.

The syntax that governs the use of these quantifiers is very similar to the syntax that governs Java for-loops.While Java for-loops loop through a collection and *perform an action* on each pass , we can think of JML quantifiers as looping through a collection and *asserting a statement* on each pass. An expression using `\forall` will be true if *every* such assertion is true, whereas an expression using `\exists` will be true if *at least one* such assertion is true.

!!! warning
Thinking of quantifiers as for-loops is helpful for learning to use them, but does not reflect how their evaluation is implemented by **OpenJML**.

Here's what our code looks like when we add our first class invariant:

PROF

```
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) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
```

```
    }

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

    public boolean isFull() {
        return size == accountIDs.length;
    }
}
```

In English, the first class invariant states that `size` never grows bigger than the length of `accountIDs`.

In English, the second class invariant states that 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 index-based way of saying that IDs must be unique.

---

4. Specify the constructor

With our class invariants in place, we're ready to specify the rest of the class. We need to provide enough information in our specifications other parts of the class that those specifications will logically imply our class invariants. **OpenJML** will use the non-class-invariant specifications to prove the class invariants. We start writing a specification for the class constructor.

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

PROF

```
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) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
    }

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

    public boolean isFull() {
        return size == accountIDs.length;
    }
}
```

Here we're specifying that all of our instance variables can be mutated using the `everything` keyword, and we're specifying that `size` is an initial value of 0 and `accountIDs` has an initial length of 100.

---

5. Specify `addAccount`

Now we move on to writing a specification of the `addAccount` method.

!!! tip
It's worth paying extra attention here. This is one of the more complicated steps in this tutorial.

To do this, we'll need to introduce some another JML feature: condtionals. The basic idea will be familiar from programming: conditionals are expressions of the form "If P then Q", and they are false if and only if "P" is true and "Q" is false. In JML they're written as "P => Q".

PROF

We have to specify enough about `addAccount` to ensure neither of our class invaraints are violated. We can do this by coming up with two conditionals that together imply our class invariants.

First Conditional: If `accountIDs` didn't already have `id` in it or `accountIDs` was full before `addAccount` executed, then after `addAccount` executed `size` is the same.

Second Conditional: If `accountIDs` did already have `id` in it and `accountIDs` was not full before `addAccount` executed, then after `addAccounts` executed `size` was incremented by 1 and `id` was stored `accountIDs[\old(size)]`.

If these conditionals seem unmotivated and complex, that's because they are. It is not immediately obvious that this is the right specificiation. The trick to seeing that it is correct, and to seeing how to come up with this kind of approach, is to follow the control flow of the method we are specifying (in this case, `addAccounts`). The only branching in our method is a single "if" block, meaning the method's behaviour is completely determined by the condition that "if" block branches on. We can therefore write two

conditionals, one for each branch of the "if" block, and show that both branches preserve our invariants. That is effectively what we have just done. The second conditional tells us what happens if enter the "if" block, and he first conditional tells us what happens if we don't. In both cases, the outcome implies that our invariants are preserved.

Now we add the JML code equivalent to our two conditionals as postconditions in our specification.

```
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;
    }

  /*@ assignable accountIDs[*], size;
    @ ensures (\old(contains(id)) || \old(isFull())) ==>
    @      size == \old(size); //
    @ ensures (!\old(contains(id)) && !\old(isFull())) ==>
    @      size == \old(size) + 1
    @   && accountIDs[\old(size)] == id;
    @*/
    public void addAccount(int id) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
    }

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

    public boolean isFull() {
        return size == accountIDs.length;
    }
```

```
}
```

6. Specify contains method

`contains` is a pure method, so we would not need to provide a specification for it (other than marking it as pure) if it were not for the fact that it is used in other specifications. However, because other specifications depend on its behaviour, we have to make that behaviour explicit by providing its own specification. Until now, we've been specifying how methods affect instance variables, but this is the first time we have had to specify a method's return value. We can do this using the `\result` keyword in an expression of the form `ensures \result == VALUE`, which expresses a postcondition that the method returns the value `VALUE`.

Using this keyword and a quantifier, it is straightforward to specify the appropriate postcondition for `contains`: `ensures \result == (\exists int k; 0 <= k && k < size; accountIDs[k] == id)`.

We're not finished specifying this method, though, because it contains a loop. The behaviour of loops is difficult for program verification tools like **OpenJML** to reason about, so we need to add additional, internal specifications called "loop invariants" to assist **OpenJML** in verifying the method specification as a whole. A loop invariant is fact that is true the beggining and at the end of every pass through a loop. Since `contains` mostly consists of a for-loop, if we can find a loop invariant of that for loop that implies our postcondition, **OpenJML** will be able to verify our specification.

We have such a loop invariant. At every iteration of our for-loop, it is true at the begining and end of the iteration that the loop index `i` is between 0 and `size` (inclusive), and also that none of values in `accountIDs` at indices between 0 and `i` are equal to `id`. If there is some value of `i` at which `accountIDs[i]` equals `id`, then the the "if" block in the loop tells us that the method will return `true`, but if there is no such value, then we go through the whole loop without returning, exicting the and arriving a the line `return false;`. Using this kind of reasoning, **OpenJML** can confirm using our loop invariant that the return value of the method is `true` if and only if `id` is stored at some index in `accountIDs`.

PROF

We add the relevant JML code below.

```
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;
    }

  /*@ assignable accountIDs[*], size;
    @ ensures (\old(contains(id)) || \old(isFull())) ==>
    @      size == \old(size); //
    @ ensures (!\old(contains(id)) && !\old(isFull())) ==>
    @      size == \old(size) + 1
    @   && accountIDs[\old(size)] == id;
    @*/
    public void addAccount(int id) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
    }
    //@ 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);
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }

    public boolean isFull() {
        return size == accountIDs.length;
    }
}
```

PROF

7. Specify isFull method

After the previous step, this one is almost automatic. We mark the method as pure and then add the right post condition, which is `ensures \result <==> size == accountIDs.length`, giving us the finished code for this task.

```
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;
    }

  /*@ assignable accountIDs[*], size;
    @ ensures (\old(contains(id)) || \old(isFull())) ==>
    @      size == \old(size); //
    @ ensures (!\old(contains(id)) && !\old(isFull())) ==>
    @      size == \old(size) + 1
    @   && accountIDs[\old(size)] == id;
    @*/
    public void addAccount(int id) {
        if (size < accountIDs.length && !contains(id)) {
            accountIDs[size] = id;
            size++;
        }
    }
    //@ 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);
        for (int i = 0; i < size; i++) {
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
    /*@ ensures \result <==> size == accountIDs.length; @*/
    public /*@ pure @*/ boolean isFull() {
        return size == accountIDs.length;
    }
}
```

PROF

8. Final verification

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

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