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.

```
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 fields 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;
        }
```

```
        return false;
    }
}
```

This makes the fields visible in JML contracts and invariants.

---

3. Add a class invariant for well-formed size

We now want to express that the value of `size` is valid — it's never negative and never greater than the array length. In JML, we specify that a property of a class never changes with a **class invariant**.

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

    //@ public invariant 0 <= size && size <= accountIDs.length;
    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 is our first invariant: it ensures the `size` field always refers to a valid prefix of the array.

---

4. Add a class invariant for uniqueness

Now we specify the main property we want to hold: the list of IDs cannot contain duplicates.

Here's how we express that with a class invariant:

```
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;
    }
}
```

PROF

This introduces a new element of JML: quantifiers. We can use *forall* to make a claim about every value in a range of values, or *exists* to make a claim about the existence of a value in a range of values. We see that JML quantifiers have a syntax similar to for-loops in Java.

In english, this JML 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 the index-based way of phrasing our condition

that bank account numbers must be unique.

5. Specify 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;
    }
}
```

6. Specify `addAccount`

We specify that `addAccount` requires the array to have space, and that `id` is not already present. It updates `size`, and the new ID is placed at the end.

```
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++;
    }

    //@ 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;
    }

    public boolean contains(int id) {
        for (int i = 0; i < size; i++) {
```

```
            if (accountIDs[i] == id) return true;
        }
        return false;
    }
}
```

This protects us from overflow and from inserting duplicates — key to maintaining the uniqueness invariant.

---

7. Specify `contains`

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.

```
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;
    }
}
```

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);
    //@ 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

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) {
        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 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;
    }
}
```

PROF

9. Add loop invariants to `removeAccount`

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

- 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;
        }
        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:

PROF

- 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!