TESTING, DEBUGGING AND VERIFICATION: DAFNY TIPS'N'TRICKS FOR JAVA PROGRAMMERS

MOA JOHANSSON

For an introductory tutorial on Dafny, see
`http://rise4fun.com/Dafny/tutorial/guide`.

This document is merely supposed to be a complement, a FAQ, to help students at the Testing, Debugging and Verification course getting started with Dafny, if used to programming in Java. It lists some aspects in which Dafny and Java differs, and some 'quirks' of Dafny that might at first seem peculiar as well as some things not covered in the tutorial. It is however not a complete list. Please contact me if you feel that there is something that should be added that might be useful to your fellow students.

1. Basics

Built in types. Dafny does not have as many built-in types as Java. It is a new language still evolving, so please see the Dafny documentation for what is available in the latest version. However, as this course is about learning how to write specifications in the first hand, and not so much about programming we will probably mostly need to use the built in types `int`, `bool`, `array<T>` and `set<T>`.

Declaring Objects. Objects are declared with keyword `new` as in Java. However, there are several ways of initialising objects.

A new object `card` belonging to a class `BankCard` can be allocated as:

```
var card := new BankCard;
```

The `new` command simply allocates the object, and returns a reference to it. Initial values of its fields are arbitrary values of their respective types. Therefore, it is common to invoke an *initialisation method* immediately afterwards, which sets the fields to sensible values. For example:

```
var card := new BankCard;
card.Init(9999,123456789);
```

Alternatively, the two statements above can be merged:

```
var card := new BankCard.Init(9999,123456789);
```

Note that Dafny allows a class to have several initialisation methods, which can be used at any time, not just when a fresh object is allocated.

Dafny does however also support *constructor methods*, similar to those in Java. Constructor methods can *only* be used when allocating new objects of a class. A class can only have one constructor methods, which is declared using the *constructor* keyword (instead of `method`):

Date: November 25, 2014.

```
class BankCard{

constructor (pin : int, accNo : int) {. . .}

}
```

The syntax for allocating a new object using the constructor method of a class is similar to that of Java:

```
var card := new BankCard(9999,123456789);
```

Where to put Semicolons (and not). Lines are typically finished with semicolons in Dafny, as in Java. However in `functions` (and `predicates`), as opposed to `methods` you do *not* use semicolons. This is because functions only consist of a single statement.

Allocating Arrays. In Dafny, a one-dimensional array (here storing `int`s) of length 5 is declared as:

```
var a := new int[5];
```

Similarly, to allocate a matrix (two-dimensional array) of size 3 by 4 we write:

```
var matrix := new int[3][4];
```

The dimensions of a multidimensional array in Dafny can be accessed using the fields `Length0, Length1` and so on.

```
matrix.Length0; // This is 3
matrix.Length1; // This is 4
```

The `forall`-construct. Both Dafny and Java supports the convenient for-each loop construct. In Dafny, this is written using the keyword `forall` (the same as the quantifier ∀, somewhat confusingly). For example, to set all entries in an array `a` to 0 you might write:

```
forall(i | 0 <= i < a.Length){
a[i] := 0;
}
```

It is often easier for Dafny's program verifier to deal with `forall`-constructs, as they aren't really loops, but rather so called *parallel assignments* to the specified array entries, so you don't have to provide a loop invariant.

2. Annotations for Specifications

Notation in Modifies Clauses. In Dafny, the `modifies`-keyword specifies which part of memory may be changed by a method. This simplifies verification, as all other memory location can be assumed to remain the same as before. When referring to a field in a modifies clause, we must prefix it by a back-tick character: `

For example, supposing `pin` is an `int` field of the `BankCard` class:

```
method ResetPIN(oldPIN : int, newPIN : int)
modifies this`pin;    // Alt. modifies `pin
. . .
```

As a second example, suppose we have a class `ATM`, which has a field `insertedCard` of type `BankCard`. To refer to the `pin` field of `insertedCard` we may write:

```
method ResetPINFromATM(oldPIN : int, newPIN : int)
modifies this.insertedCard`pin;
. . .
```

The back-tick notation *only* applies to modifies clauses.

Framing and the `reads`-keyword. The *reading frame* specifies the memory locations that a *function or predicate* is allowed to read. This is again an optimisation for verification: When we write to memory, we can be sure that functions that did not read that part still have the same value. No re-checking is needed.

Reading frames are at the level of *objects* (not fields of primitive types). Suppose we have a predicate `isPinValid()` in the `BankCard` class, which need to access the `pin` field. As `reads` works on the level of object, we must however declare that this predicate reads `this`:

```
predicate isPinValid()
reads this;
{ . . . }
```

However, note that if a predicate or function needs to access *both* fields of primitive type and fields of object types this needs to declared explicitly:

```
predicate myPredicate()
reads this, this.someFieldOfObjectType;
{ . . . }
```

Just writing `reads this` is not enough, we will also need to explicitly include fields of object-type that are read.

The `old` keyword. In specifications, you commonly want to say something about how a method changes some value with respect to its old value, i.e. the value before the method was called. For this, we have the keyword `old`, which is used in `requires` clauses. For example, to add a requirement that `ResetPIN` really must change the PIN to a value different from what it was, you may write:

```
method ResetPIN(oldPIN : int, newPIN : int)
. . .
requires pin != old(pin);
. . .
```

The `fresh`-keyword. It is sometimes important for the verifier to know that some given object has been *freshly allocated* in a given method. For instance, suppose you have some class with an `array` field `a`, which the `Init` method should initialise by creating a new array object. Here, one should include the post-condition `fresh(a)`, to capture this requirement.

```
method Init (len : int)
modifies this;
. . .
ensures fresh(a);
{
   a := new int[len];
   . . .
```

```
}
```