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 (for instance, strings and chars are currently not available). For the purpose of this course, you will probably mostly need to use `int`, `bool`, `array<T>` and `set<T>`.

Declaring Objects. Objects are declared with keyword `new` as in Java, but there are no constructor methods as in Java. Instead, you first typically declare a new object, then call its `Init` method to initialise its fields. For example:

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

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.

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 $\forall$, somewhat confusingly). It 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. The notation for referring to a field in a modifies clause is different depending on whether the field is of a primitive type or an object type. Primitive types are prefixed 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;
. . .
```

This applies to any field of primitive types like `int`, `bool` or value types like `set<T>`. Array is an object-type and not affected by this quirk. Note that in code, you still refer to the fields as in Java also for primitive types, i.e. `this.pin`. The back-tick notation *only* applies to modifies clauses.

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];
   . . .
}
```