Dafny Cheatsheet

Imperative and OO

Keyword(s)	What it does	Snippet

var	declares variables	`var nish: int;`
		`var m := 5;  /* inferred type */`
		`var i: int, j: nat;`
		`var x, y, z: bool := 1, 2, true;`

:=	assignment	`z := false;` `x, y := x+y, x-y;  /* parallel assignment */`
if..else	conditional statement	`if z { x := x + 1; }  /* braces are */` `else { y := y - 1; }  /* mandatory */`
if..then ..else	conditional expression	`m := if x < y then x else y;`
while forall	loops	`while x > y { x := x - y; }` `forall i | 0 <= i < m { Foo(i); }`

method returns	subroutines	`/* Without a return value */`
		`method Hello() { print "Hello Dafny"; }`
		`/* With a return value */`
		`method Norm2(x: real, y: real)`
		`  returns (z: real)  /* return values */`
		`{  /* must be named */`
		`  z := x * x + y * y;`
		`}`
		`/* Multiple return values */`
		`method Prod(x: int) returns (dbl: int, trpl: int)`
		`{ dbl, trpl := x * 2, x * 3; }`

class	object classes	`class Point  /* classes contain */`
		`{  /* variables and methods */`
		`  var x: real, y: real`
		`  method Dist2(that: Point) returns (z: real)`
		`  requires that != null`
		`  { z := Norm2(x - that.x, y - that.y); }`
		`}`

array	typed arrays	`var a := new bool[2];`
		`a[0], a[1] := true, false;`
		`method Find(a: array<int>, v: int)`
		`  returns (index: int)`

Specification

Keyword(s)	What it does	Snippet

requires	precondition	`method Rot90(p: Point) returns (q: Point)`
		`  requires p != null`
		`{ q := new Point; q.x, q.y := -p.y, p.x; }`

ensures	postcondition	`method max(a: nat, b: nat) returns (m: nat)`
		`  ensures m >= a  /* can have as many */`
		`  ensures m >= b  /* as you like */`
		`{ if a > b { m := a; } else { m := b; } }`

assert **assume**	inline propositions	`assume x > 1;` `assert 2 * x + x / x > 3;`

! && ||
==> <==
<==>
	logical connectives	`assume (z || !z) && x > y;`
		`assert j < a.Length ==> a[j]*a[j] >= 0;`
		`assert !(a && b) <==> !a || !b;`

forall **exists**	logical quantifiers	`assume forall n: nat :: n >= 0;` `assert forall k :: k + 1 > k;  /* inferred k:int */`

function
predicate
	pure definitions	`function min(a: nat, b: nat): nat`
		`{  /* body must be an expression */`
		`  if a < b then a else b`
		`}`
		`predicate win(a: array<int>, j: int)`
		`  requires a != null`
		`{  /* just like function(...): bool */`
		`  0 <= j < a.Length`
		`}`

modifies	framing (for methods)	`method Reverse(a: array<int>)  /* not allowed to */` `  modifies a  /* assign to "a" otherwise */`
reads	framing (for functions)	`predicate Sorted(a: array<int>)  /* not allowed to */` `  reads a  /* refer to "a[_]" otherwise */`

invariant	loop invariants	`i := 0;`
		`while i < a.Length`
		`  invariant 0 <= i <= a.Length`
		`  invariant forall k :: 0 <= k < i ==> a[k] == 0`
		`{  a[i], i := 0, i + 1;  }`
		`assert forall k :: 0 <= k < a.Length ==> a[k] == 0;`

set
seq
multiset
	standard data types	`var s: set<int> := {4, 2};`
		`assert 2 in s && 3 !in s;`
		`var q: seq<int> := [1, 4, 9, 16, 25];`
		`assert q[2] + q[3] == q[4];`
		`assert forall k :: k in s ==> k*k in q[1..];`
		`var t: multiset<bool> := multiset{true, true};`
		`assert t - multiset{true} != multiset{};`
		`/* more at: */`
		`/* http://rise4fun.com/Dafny/tutorial/Collections */`