

Dafny Cheatsheet

Imperative and OO

Keyword(s)	What it does	Snippet
var	declares variables	<pre>var nish: int; var m := 5; /* inferred type */ var i: int, j: nat; var x, y, z: bool := 1, 2, true;</pre>
:=	assignment	<pre>z := false; x, y := x+y, x-y; /* parallel assignment */</pre>
if..else	conditional statement	<pre>if z { x := x + 1; } /* braces are */ else { y := y - 1; } /* mandatory */</pre>
if..then ..else	conditional expression	<pre>m := if x < y then x else y;</pre>
while forall	loops	<pre>while x > y { x := x - y; } forall i 0 <= i < m { Foo(i); }</pre>
method returns	subroutines	<pre>/* 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; }</pre>
class	object classes	<pre>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); } }</pre>
array	typed arrays	<pre>var a := new bool[2]; a[0], a[1] := true, false; method Find(a: array<int>, v: int) returns (index: int)</pre>

Specification

Keyword(s)	What it does	Snippet
requires	precondition	<code>method Rot90(p: Point) returns (q: Point) requires p != null { q := new Point; q.x, q.y := -p.y, p.x; }</code>
ensures	postcondition	<code>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; } }</code>
assert assume	inline propositions	<code>assume x > 1; assert 2 * x + x / x > 3;</code>
! && ==> <== <==>	logical connectives	<code>assume (z !z) && x > y; assert j < a.Length ==> a[j]*a[j] >= 0; assert !(a && b) <==> !a !b;</code>
forall exists	logical quantifiers	<code>assume forall n: nat :: n >= 0; assert forall k :: k + 1 > k; /* inferred k:int */</code>
function predicate	pure definitions	<code>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 }</code>
modifies	framing (for methods)	<code>method Reverse(a: array<int>) /* not allowed to */ modifies a /* assign to elements of "a" otherwise */</code>
reads	framing (for functions)	<code>predicate Sorted(a: array<int>) /* not allowed to */ reads a /* refer to "a[_]" otherwise */</code>
invariant	loop invariants	<code>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;</code>
set seq multiset	standard data types	<code>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 */</code>