CITS5501 Software Testing and Quality Assurance
Program verification

Unit coordinator: Arran Stewart

Overview

- Why verify programs?
- How can we verify programs using Dafny?

Program verification

Program verification (also called "formal verification") is the process of **proving** that a program satisfies a formal specification of its behavior.

Motivation

Why formally verify a program?

Motivation

Why formally verify a program?

Because you want to be *absolutely* sure it meets (some or all of) its specifications.

Limitations

Why not formally verify all programs?

- It does take extra time, effort and expertise
  - Though as we will see, languages like Dafny make formal verification much easier than it once was
- It doesn't guarantee that our specifications were sensible ones
- Appropriate tools might not be available for the language you're working in

How does verification work?

- Proofs of correctness use techniques from formal logic to prove that if the starting state (i.e., "input" variables) of a program satisfies particular properties, than the end state after executing a program (i.e., "output" variables) satisfies some other properties.
  - The first lot of properties are called *preconditions* (assertions that hold prior to execution of a piece of code) –
  - The second lot are *postconditions* (assertions that hold after execution)

Preconditions and postconditions review

- We've seen that in all languages, it's good practice to document the preconditions and postconditions of a method or function.
- The preconditions and postconditions are like a *contract* between the developer of the method, and the caller of the method;
  - *If* the preconditions are satisfied by the caller, then the method promises that after execution, the postconditions will be true.
  - But if the preconditions are *not* satisfied, the method doesn't promise anything at all.

Undefined behaviour example

- See for example the Java `Arrays.binarySearch(arr, key)`, methods which say "The array must be sorted ... prior to making this call. If it is not sorted, the results are undefined."
- If the caller passes an unsorted array – the method implementer can do *anything they want*.
  - They can return a wrong answer; delete all the files on your hard drive; launch intercontinental ballistic missiles at your house.

Postconditions and exceptions

In Java, throwing exceptions is just another sort of postcondition.

e.g. a method which opens a file on disk may have the contract:

> "If the file exists, and is readable, I will return a properly constructed `FileInputStream` object.
> If the named file does not exist, is a directory rather than a file, or for some other reason cannot be opened for reading, I will throw a `FileNotFoundException`."

(See the constructor description for the class `java.io.FileInputStream`.)

Dafny

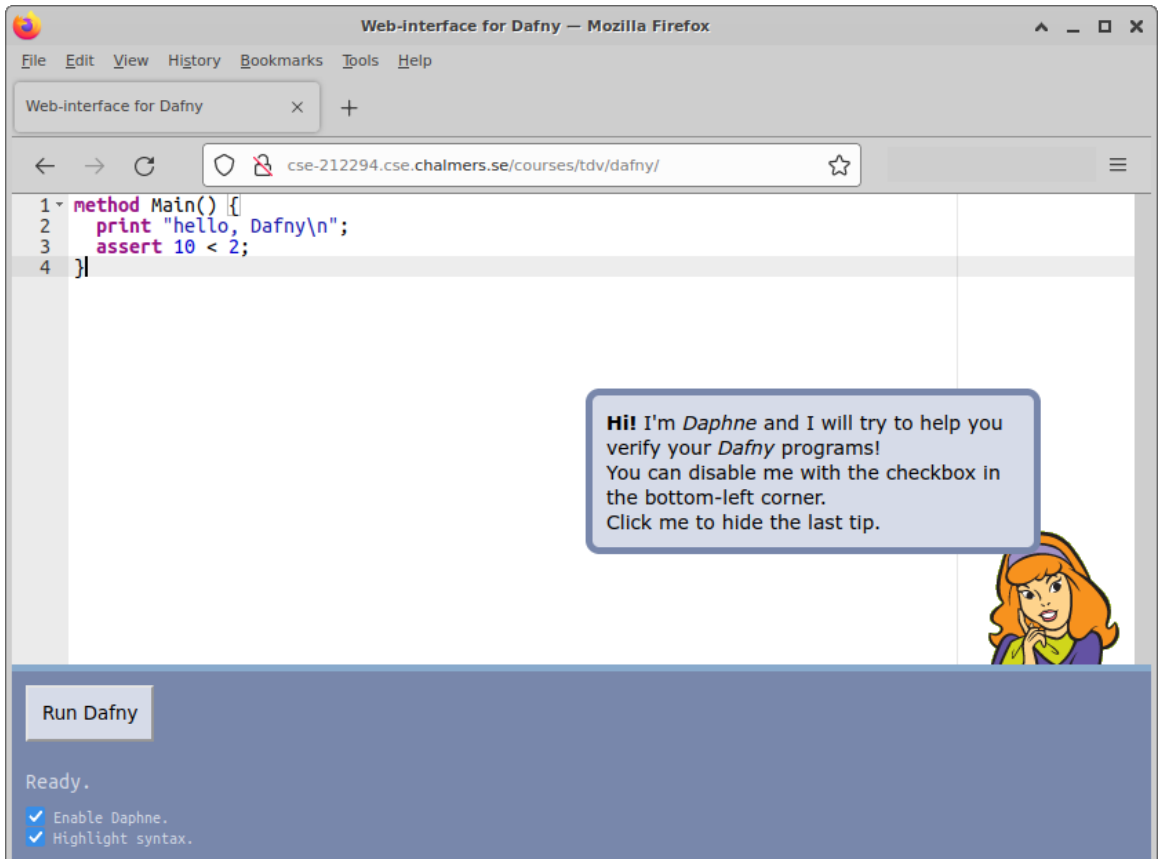
We'll be using the Dafny programming language to explore program verification.

It is somewhat similar in style to Java or C#, but includes built-in features for program verification.

Once a program is verified in Dafny, it can be compile to C#, Java, JavaScript or Go.

Using Dafny online – Chalmers

Chalmers University of Technology, Sweden, provides a web page where you can compile Dafny code.



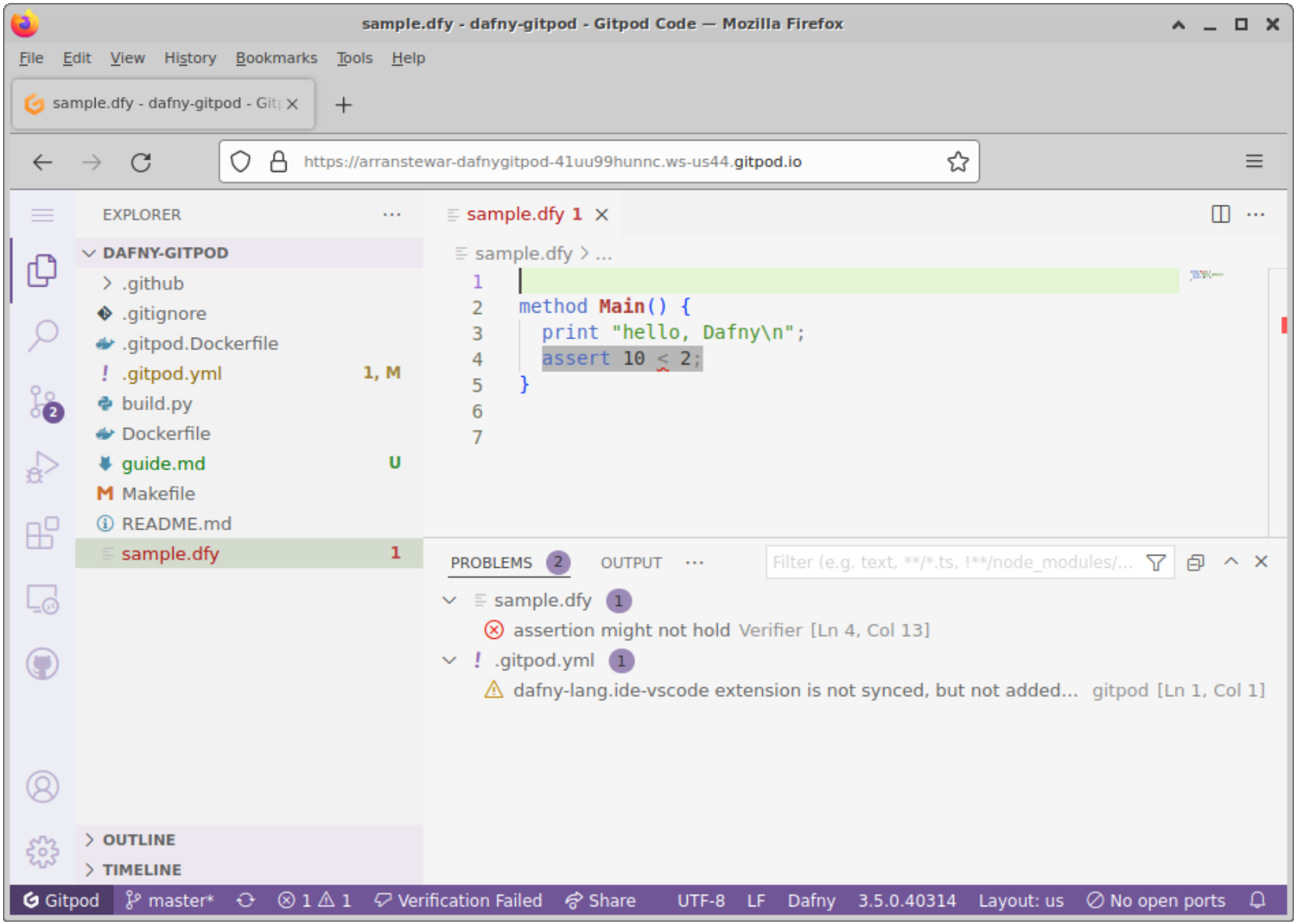
http://cse-212294.cse.chalmers.se/courses/tdv/dafny/

Using Dafny online – Gitpod

You can also compile (as well as run) Dafny programs using Gitpod, which provides you with an online development environment based on Microsoft VS Code.

- Visit https://gitpod.io/#https://github.com/arranstewart-dev/dafny-gitpod in your browser
- Gitpod will download a Docker image containing the Dafny compiler, and create an online IDE environment
  (This may take a couple of minutes)
- In the terminal shell, the "`dafny`" command will be on your `PATH`.

Dafny in Gitpod



Dafny methods

To write and use a method `Abs()` which calculates the absolute value of an integer, we would write code something like this:

```
method Abs(x: int) returns (y: int) {
   if x < 0
      { return -x; }
   else
      { return x; }
}

method Main() {
  var x := Abs(-3);
  print x, "\n";
}
```

Abs method

This method doesn't (yet) include any preconditions or postconditions.

```
method Abs(x: int) returns (y: int) {
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Dafny return values

One difference from Java is that the return value is given its own name, "y".

```
method Abs(x: int) returns (y: int) {
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Why is this? It's because we can add postconditions to Dafny code, which refer to the return value (or to input parameters, as well), so it's convenient to give it a name.

A common convention is that if the input parameter is called `x`, we call the return value `x'`.

Dafny preconditions and postconditions

We can add preconditions (with "`requires`") and postconditions (with "`ensures`") to a method – and Dafny *won't compile* the method, unless we can convince the compiler that *if* the preconditions are satisfied, *then* all the postconditions follow.

```
method Abs(x: int) returns (y: int)
  ensures y >= 0
{
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Dafny knows enough about arithmetic and negative numbers that it can assure itself that `y >=0` is always true.

Dafny preconditions and postconditions

We don't *have* to include any preconditions or postconditions at all; but being able to do so is the main point of using Dafny.

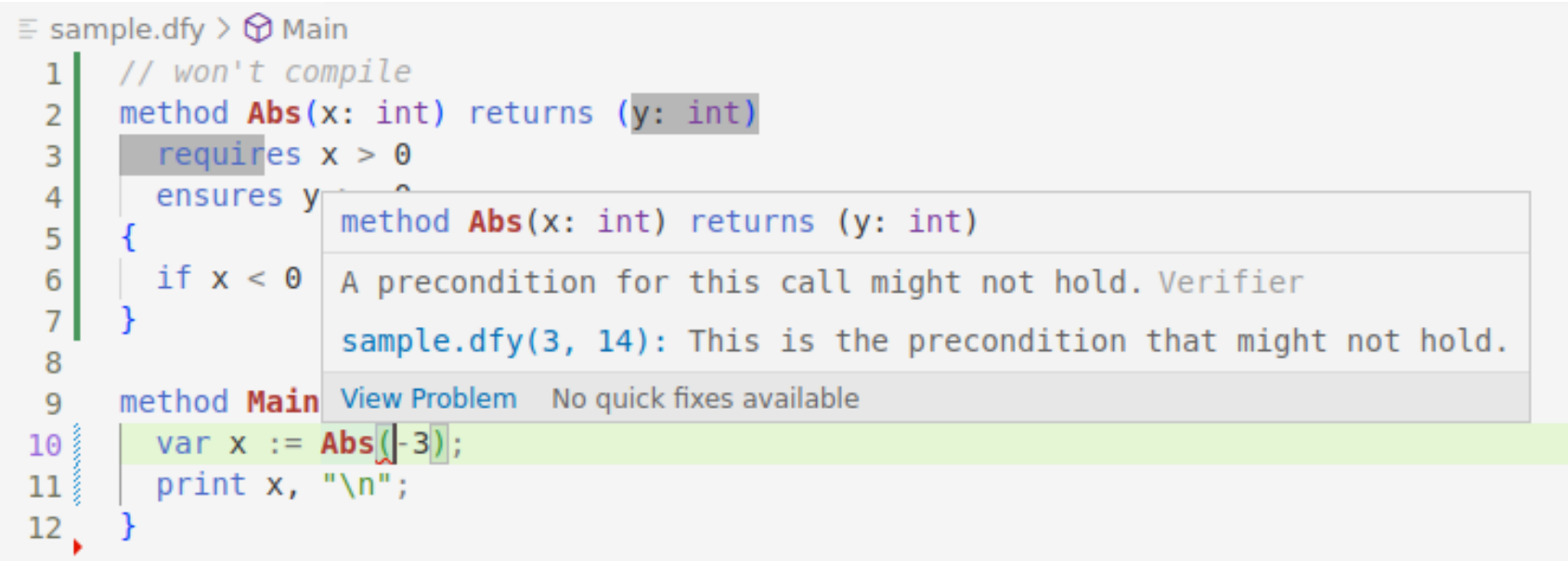
Dafny won't let us *call* a method, unless we can prove we've satisfied the preconditions for that method. If we can't, we get a compile error.

```
// compiles ok
method Abs(x: int) returns (y: int)
  ensures y >= 0
{
  if x < 0 { return -x; } else {return x; }
}

method Main() {
  var x := Abs(-3);
  print x, "\n";
}
```

Dafny preconditions and postconditions

```
// won't compile
method Abs(x: int) returns (y: int)
  requires x > 0
  ensures y >= 0
{
  if x < 0 { return -x; } else {return x; }
}

method Main() {
  var x := Abs(-3);
  print x, "\n";
}
```

Dafny preconditions and postconditions



Dafny postconditions

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y
{
   ...
}
```

- Multiple "ensures" specifications can be added
- "ensures" specifications can make use of the ususal logical connectives (e.g. "`&&`", "`||`")
- The suggested style is for distinct "properties" to be given their own "ensures" specification
- A method with no "`ensures`" specifications has no postconditions, so will always verify.

Dafny preconditions

Preconditions can be specified with keyword "`requires`"

```
method AddOne(x: int) returns (y: int)
   requires x > 0
   ensures  y > 0
{
   return x + 1;
}
```

Summary

- A programmer *calling* a method must ensure the preconditions are met
  (else Dafny reports a compile error)
- A programmer *writing* a method may assume the preconditions are already true, but must ensure the postconditions are met
  (else Dafny reports a compile error)

Dafny assertions

In addition to preconditions and postconditions, Dafny lets you write *assertions* – these are found somewhere in the body of a method.

They assert that something is true at that point in the code (and if Dafny can't prove it is so, it will report an error).

Dafny assertions

```
method MyMethod()
{
  assert 2 < 3;
}
```

Assertions don't *have* to mention any of the variables or return values of a method (though obviously they are going to be more useful if they do).

Dafny assertions

You can think of assertions as a way of "asking" the Dafny verifier what it knows to be true at any point in the program.

(A bit like `println`/`printf`, but for debugging *compilation*, rather than *execution*.)

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y
{
   if x < 0
      { return -x; }
   else
      { return x; }
}
method MyMethod() {
  var v := Abs(-3);
  assert v >= 0;
}
```

Dafny verification errors

- There are two main reasons you might get a verification error:
  - Firstly, there might be something actually incorrect with your code.
  - Secondly, it might be correct, but the Dafny verifier isn't "clever" enough to prove that the required properties hold.
- In the latter case, we need to give it some help. We'll see an example.

Proving loops correct

Loops pose a problem for Dafny.

When the verifier sees a method containing a loop, it doesn't know in advance how many times the loop will be executed. There are potentially infinite paths through the program.

And to prove that the postconditions are true (assuming the preconditions are), the verifier needs to consider *all* the possible paths through a method.

Loop invariants

The solution is to make use of *loop invariants*.

These are expressions that hold true

- before entering the loop
- after every execution of the loop body

(but they may be false at various points in the middle of the loop body).

Loop invariant example

Loop invariants are put just before the body of a loop:

```
var i := 0;
while i < n
   invariant 0 <= i
{
   i := i + 1;
}
```

Loop invariant example

```
var i := 0;
while i < n
   invariant 0 <= i
{
   i := i + 1;
}
```

The verifier reasons as follows:

- Is `0 <= i` true before the loop starts?
  - Yes, since `i` is 0, and `0 <= 0` is true.
- *If* the invariant was true at the start of the loop, will it also be true at the end of the loop?
  - Yes, it will.
    If `0 <= i` at the start of the loop, all we do in the body is increment `i` by 1; so `0 <= i` will *still* be true at the end of the loop.
- From this, Dafny concludes that if the invariant was true *before* entering the loop, it will also be true *after* the loop (since there's no place it could have been made false)

Loop invariant applications

The example above is very simple, but we can work our way up to more complex loops.

For instance, here is a loop that calculates $m \times n$ (though in any modern programming language, we already have integer multiplication):

```
// assume m and n are parameters, say
var tot := 0;
while m > 0
{
  tot := tot + n;
  m := m − 1;
}
```

Could we *prove* that, after the loop ends, $tot = m \times n$?

Loop invariant applications

It makes things easier if, rather than altering `m` and `n`, we leave them as is and copy their values into other variables. Let's write this as a method in Dafny.

(In fact, Dafny will not *let* us mutate parameters.)

```
method MyMethod(m : int, n : int) {
var tot := 0;
var a := m; var b := n;
while a > 0
  {
    tot := tot + b;
    a := a − 1;
  }
}
```

Loop invariant applications

Now we can write a postcondition in terms of m and n:

```
method MyMethod(m : int, n : int) returns (r: int)
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    {
      tot := tot + b;
      a := a - 1;
    }
  return tot;
}
```

This will fail, as Dafny cannot prove it is true.

Loop invariant applications

One thing that is always true about the loop:

- `tot` is the "total so far"
- If we add the bits "still to go" (`a * b`) to the total, we should get `m * n`.

```
method MyMethod(m : int, n : int)
  returns (r: int)
  ensures r == m * n
{ // won't compile
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
  return tot;
}
```

So an invariant is `a * b + tot == m * n`.

Loop invariant applications

That won't compile, because we forgot to consider that `m` might be negative. If it were, we'd end up with an endless loop.

So let's make sure `m` and `n` are non-negative.

```
method MyMethod(m : int, n : int) returns (r: int)
  requires m >= 0 && n >= 0
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
  return tot;
}
```

Loop invariant applications

```
method MyMethod(m : int, n : int) returns (r: int)
  requires m >= 0 && n >= 0
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a — 1;
    }
  assert tot == m * n;
  return tot;
}
```

Dafny will confirm that this method is correct – it understands enough basic arithmetic to work out that the loop invariant holds before and after each loop iteration.

Loop invariant applications

```
  // ...
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
}
```

And if the loop invariant holds in those cases, it also holds after; and since `a == 0` after the loop,

```
   a * b + tot == m * n
→ 0 * b + tot == m * n
→ 0 + tot == m * n

→ tot == m * n
```

Compilation

Once the program is verified, we can compile it into various formats.
The default format is a Windows library, but we can also compile to (say) Java or C++.

sample.dfy

```
method Main() {
  var result := MyMethod(3, 5);
  print "result ", result, "\n";
}
method MyMethod(m : int, n : int)
  returns (r: int)
  requires m >= 0 && n >= 0
  ensures r == m * n
{ var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
  return tot;
}
```

Compilation

```
$ dafny /compileTarget:java sample.dfy
$ java -cp /opt/dafny/DafnyRuntime.jar:./sample-java sample
Picked up JAVA_TOOL_OPTIONS:  -Xmx3435m
result 15
```

Power of specifications

Although these are only small examples, hopefully you can see that this technique is quite powerful.

If we can prove that small portions of code are correct (i.e., meet their specification), and we can chain them together, then we will be able to prove correctness of large programs.

Example assertions

We can use postconditions, preconditions, assertions and invariants to express:

- Bounds on elements of the data:

  $$n \geq 0$$

- Ordering properties of the data:

  $$\text{for all } j : 0 \leq j < n - 1 : a_j \leq a_{j+1}$$

- "Finding the maximum"

  e.g. Asserting that p is the position of the maximum element in some array $a[0..n-1]$

  $$0 \leq p < n \vee (\text{for all } j : 0 \leq j < n : a_j \leq a_p)$$