Lecture 1

The first bug ever in programming was discovered from a program written by grace hopper

- An actual bug (moth) was in a relay system which threw off the code
- This is where the name bug comes from
- When some unexpected behavior occurs

Software Crisis

In 1968 there was a conference held in Germany by NATO

- Top 50 computer scientists, programmers and industry leaders grouped together to discuss the state of programming
- In this conference, this is where the term “Software Engineering” came from
- The term was to signify a **systematic, disciplined, quantifiable approach to the production and maintenance of software**

Large Software systems often don’t provide their desired functionality based on their requirements

- 75% of large systems do not provide the required functionality
- Requirements are dynamic and based on many external factors, and change before, during and after development.

Poor time management and cost estimation in development are an organizational “nightmare” and generally there are severe budget overruns

- 33% of large projects are never completed (lots of cancellation)
- Most large projects overshoot their time schedule by half
- In a lot of cases going 2-5x over your budget is considered okay and normal
- In some cases, people go 10x over their budget

We want to be able to model software, but this can be tricky, since we have infinite state spaces, and many factors that can throw unexpected behavior. To be able to correctly model our code we need to have the range of variables in each state and be able to prove within that range, the program will behave correctly. This is the only way we can model behavior.

Problems with software

Complexity in Software Engineering

Software is complex because of

- Irregularities
    - Hardware is regular (static and not changing)
    - Hardware engineering is systematic
- Infinite and discrete state space with no physical laws of abstraction
    - Characteristics of hardware can be tested in isolation because it is physical (strength of bridge, stiffness of metal etc.)
    - Very difficult to abstract behavior of software since it is conceptual since software is abstracted
        - Software is just us observing the side effects of our code behavior.
- Very difficult to test, since there is an infinite and discrete state space.
- Any variable can break a program even after being extensively tested (divide by 0, null, integer over/underflow etc.)

For example in civil engineering, if you test a bridge with 2 trains and 100 cars, you know that for all vehicles below 100 cars and 2 trains it works, in programming there is no such concept, you can change a variable which can modify another variable out of scope and begin undefined behavior.

We don't run dafny programs, dafny is just a way to prove that a certain set of logical instructions provide the correct behavior SPECIFIED. We need to give specifications (invariants/pre and post conditions) and provide code that performs behavior on our variables, and dafny will try and prove us wrong with induction. If it can't your all good! But if it can it will throw you back an error.

Conformity in software engineering

- Software lives in a volatile environment of users and specifications and hardware that are consistently changing
    - Software conforms to hardware, software must run on the hardware
    - Software conforms to the specifications provided and requirements (typically)
- Software must adapt
    - It must keep up with hardware and people.
- This creates a tricky problem of handling change, do we re-use existing code we think is correct or do we begin from scratch?

Flexibility of software engineering

The idea of software being easily changed leads to

- Frequent requests to add new features
- Frequent requests to modify existing systems and tweak
- Frequent requests to support new hardware
- This can also lead to the problem of unmaintainable code

- Even at work I was asked to modify front end elements almost daily
- Requests can come in these timeframes
    - During development (change request)
    - After development (retro-requirements)
    - Software engineers must be able to predict what the user will want in the future and where the future growth will be
    - Software engineers must also do this externally on their own, as it is usually not a requirement from a business perspective.

Reliability in software engineering

There is only one Sydney harbor bridge, there is no backup, if it goes down there is a huge problem. Generally, in programming something goes wrong, we just say turn it on and off again, there is no turning on and off the harbor bridge.

Software is considered naturally bug prone, unlike hardware where it rarely occurs.

Example of difference between how bug prone hardware and software is

- Pentium bug in 1995 (Hardware)
    - Effected Floating point unit calculations (FPU) in intel Pentium processors
    - The processor could return incorrect result from dividing floating point numbers
    - In production for a year before anyone noticed
    - Major scandal with huge backlash
    - Cost $475m in replacements.
- Apple and Microsoft OS (Software)
    - Removed by regular and free updates
    - Bugs are accepted and expected
    - Cost is non-transparent
    - Companies need to have certain ways to deal with bugs in existing software

The Pentium Bug

According to the Pentiums that were bugged, (4195835/3145727) = 1.3338 when the answer was 1.3337

The numerator is stored in the hexadecimal pattern 0x4005FB where the number 5 caused a fault in the control logic of the FPU. This FPU was used when any calculation occurred, and the fault caused many controversies. Intel claimed it happened once every 27,000 years, but IBM called them out and said BS, it can occur once in 24 days.

Hardware problems have much more serious consequences.

Abstractions in Software

- You cannot see software, you only see the side-effects
    - It's impossible to understand large software systems at a base level, it is far too complex
- There is no physical model that can help view and define it
    - A blueprint of a building helps both the architect and client understand what the building will look like
    - CAD systems can visualize bridges, complex structures
    - Best thing we can do in software is create flow charts/ER diagrams/use cases/scenarios
        - Extremely difficult
        - Narrow views of behavior
    - Even if we unit tested the core components it is still difficult to understand how they will all work together

Adaptability in Software

- New technology and advances in hardware increases the demand on software
    - Need to write a compiler for latest and new chips
- New uses such as cloud computing, social networks, search engines etc. revolutionize the use of software.
- Guaranteed behavior, some mission critical systems such as performing surgery with machines, military security, all demand correct behavior with no faults as the consequences would be catastrophic.

What can we learn from Arianespace?

Arianespace is the world's largest commercial satellite launch company

- Founded in 1980, now a multi-billion-dollar company
- Launches from French Guiana
- Launched over 600 satellites in 38 years
- Different and newer versions of launch vehicles
    - Ariane 1-5
    - Ariane 6 scheduled to launch in 2020

The Ariane 5

The Ariane 5

- Weighed 800 tons
- Heavy-lift launch vehicles, carries 10 tons of satellites
- Launched around 100 missions to date, 14 planned in 2018
- Has a reputation for reliability
    - First launch in 1996 was a total failure, the rockets onboard computer said it was off track by 90 degrees and tried to correct it resulting in massive forces requiring manual detonation
    - First successful launch in 1997
    - Another total launch failure in 2002

Why did the Ariane 5 have a launch total failure in 1996?

Guidance system sent incorrect information. As a result, the rocket disintegrated costing half a billion dollars

- Software in the rocket used a mix of 64-bit and 16-bit technology. For integer data conversion the 16-bit limit was 32768, whereas the 64-bit limit is over 2 billion. This means converting from 64 bit to 16-bit numbers could cause undefined behavior.
- The designer chose to just catch the exception as opposed to leaving it alone (the code executed will halt and throw an error if not)
- This caught error was sent to the onboard computer as a literal string, whereas the computer just ran expecting numbers on its coordinates and basis this meant the computer just converted the literal string into numbers.
- This offset the guidance system, to believe it was going 90 degrees in the wrong direction, in where it tried to correct causing the rocket to tilt sideways rapidly and explode.

This occurred since code from the Ariane 4 was the 16-bit technology that was assumed correct. This is because the Ariane 4 was already working, no one expected anything to go wrong using correctly existing code. So, it was just plugged in and not even tested. The Ariane 5 was a lot faster than earlier designs, so the speeds went over the 16-bit limit.

What can we learn? (favorite question used in 2 exams)

What was the lesson learnt from the Ariane 5 space disaster?

- Code Reuse
    - Ariane 4 was successful, so code was trusted and assumed correct and thus re-used in Ariane 5 without testing
- Systems/integration testing was incomplete
- There were design faults
    - Don’t just bloody throw exceptions, never ignore them
    - It should never be the case that a message is read as data, that’s just ridiculous
- Designers made naïve and dangerous assumptions
    - Ariane 4 designers assumed that the data won’t be over 16-bit, and nothing would ever occur
    - Ariane 5 designers assumed reused code was correct and didn’t bother to test.

Why do we always need testing?

- Even after we have verified the code is correct, we must test to see if the code is fit for its purpose and does the right thing.
- Your code can do bubble sort correctly, good for you that’s what you want and you can test and make sure that’s correct, but if you don’t need sorted data, then you haven’t really tested any code in respect to the requirements.

Lecture 2

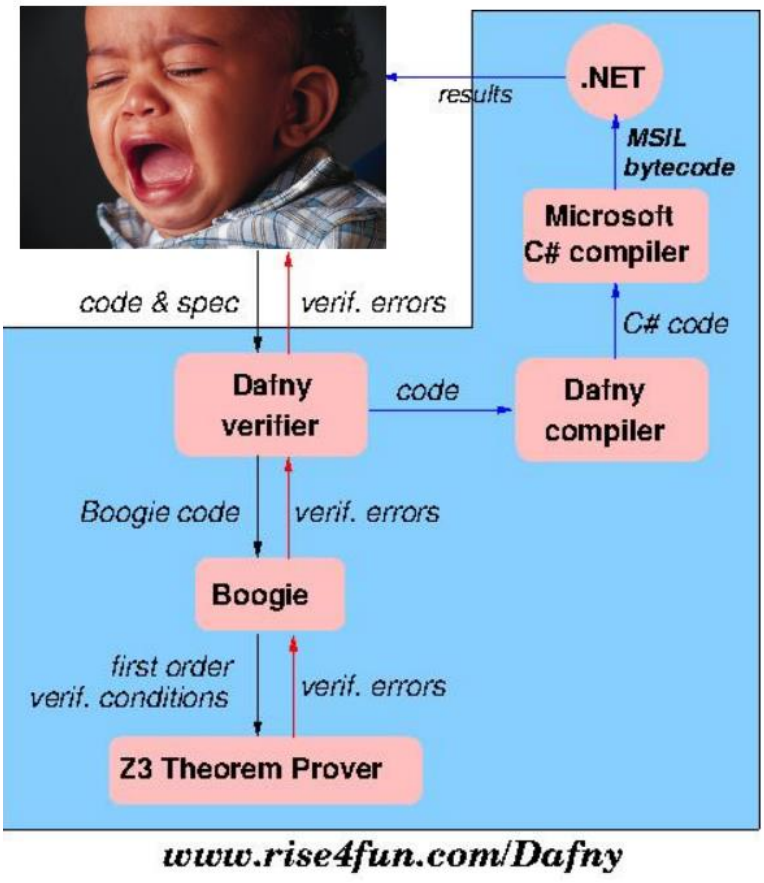
Dafny was developed by Microsoft research in a team of 15 containing Rustan Leino who has now moved on to senior principal engineer in AWS.

Dafny is a new generation of language that can address the problem of correctness.

Dafny is

- Cutting edge
- A Microsoft research product, not a production system
- An integrated programming and reasoning language
    - Programming language is imperative, object-based
        - Translated into C#
        - Translated down into .NET and JavaScript and Go
    - Reasoning language is functional and acts as a static verifier
        - Translated to Boogie which acts as the backend
        - Boogie interacts with the Z3 SMT solver

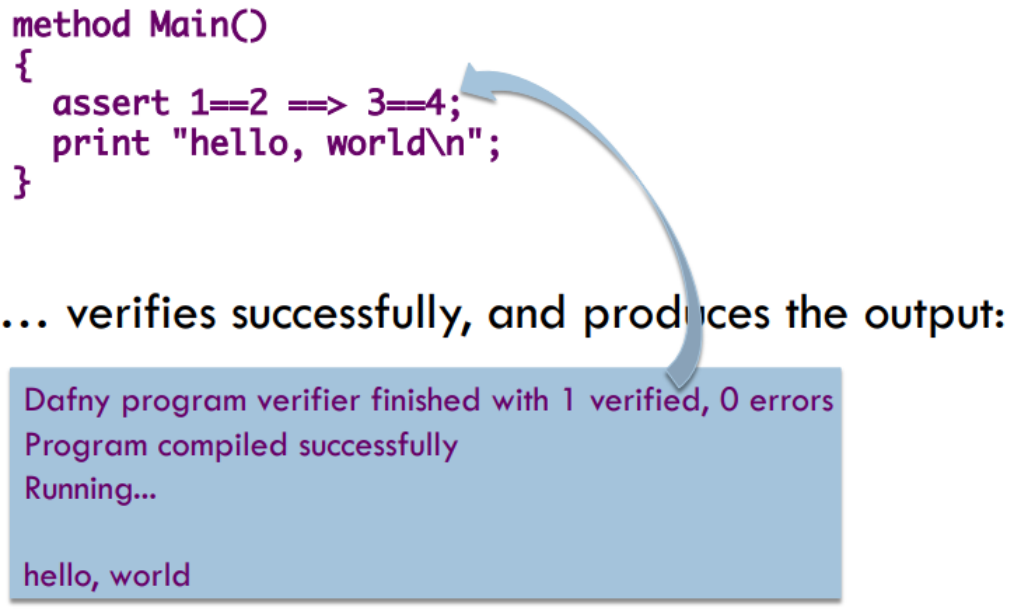
The basic flow of Dafny follows



User provides the code and specifications to the Dafny verifier. Dafny verifier will pass it down to boogie which will interact with Z3 to verify the code to the specifications. Any errors are sent up the chain to

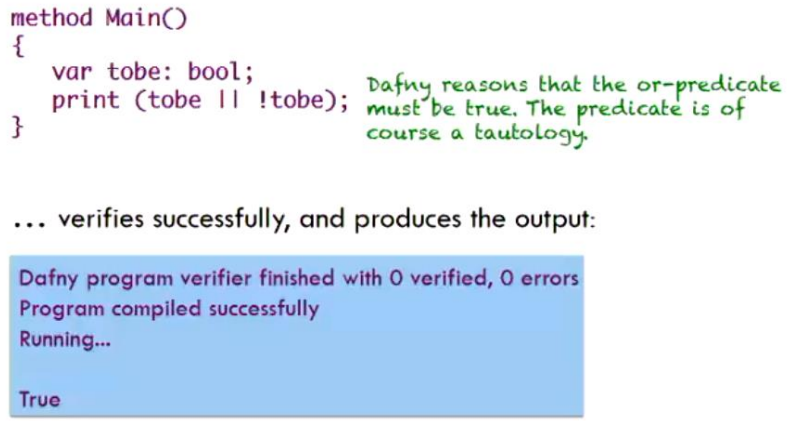
dafny. If it is correct the verifier will pass it through the dafny compiler which translates into c# code and the result is sent back to the user.

A simple example of hello world in Dafny which includes forced verification



This program first verifies using an assert that the Boolean is indeed correct. In this case we are saying that 1 is equal to 2, implies 3 is equal to 4. This is correct since from any false conclusion we can assume anything (explosion rule), if there are two Mondays, I will grow 300 cm, there will never be two Mondays. False implies False. These asserts are passed through a truth table to then verify the correctness of the assertion.

Another example of Dafny



In this case we are printing the result of True or Not true, which is a tautology (universe) and will always be true. Even though we did not initialize tobe, Dafny doesn't care, since it will check for all values of the Boolean using a truth table. For all values of tobe, it will always be true.

Dafny will not compile a program that cannot be verified.

The program

```
method Main()
{
  var i;
  assert i!=123456789;
  print "you will never see me";
}
```

Notice Dafny can infer types

... fails to verify

```
Dafny 2.1.1.10209
stdin.dfy(4,11): Error: assertion violation

Dafny program verifier finished with 0 verified, 1 error
```

Since we did not initialize I, dafny does not know what I can be, and there is a possibility that I can be 123456789, so it will say, in that case WRONG so for all values of I it is not true. And will fail.

The verification is the on/off switch for running, any violated assertions = no run. However, this does mean you can put unverified code (not recommended). Dafny also uses duck-typing.

We can also perform some Boolean and propositional logic in Dafny

```
method Tautology()
{  var p, q;
   assert (p==>q)||(q==>p); // a tautology
}
Dafny verifier program finished with 1 verified, 0 errors
```

Obviously bools

Or in a much longer approach

```
method TruthTable()
{
   var p, q;
   p:=true;  q:=true;  assert (p==>q)||(q==>p);
   p:=true;  q:=false; assert (p==>q)||(q==>p);
   p:=false; q:=true;  assert (p==>q)||(q==>p);
   p:=false; q:=false; assert (p==>q)||(q==>p);
}
Dafny verifier program finished with 1 verified, 0 errors
```

As can be seen, both are the exact same piece of code, but dafny is smart enough to translate the first into the second.

In any programming language, when a run-time error occurs, the program crashes, anything after it is not executed (the scattered print statements before the crash). Assert statements, are like creating the crash to halt execution upon failing of assertion which is a software-based crash like SIGINT. Asserts should happen at compile time not runtime which is the case in C.

Potentially examinable

- start

A dafny assert is a real assert, which will assert for the range of the variable so it will create a compile time error, whereas C assert, will just create a crash when it comes across that value. Asserts should hold before run-time, runtime crashes forced manually by C asserts are ridiculous. Learn the difference between asserts in C and asserts in dafny.

- end

Dafny is also capable of performing induction.

```
method main() {
  var i;
  assert i < i+1; // i is not defined
}
Dafny verifier program finished with 1 verified, 0 errors
```

With induction, this proof becomes very simple. Even though I is undefined, for all numeric values of I, this statement will always hold. There is no concept here of antialiasing and checking the values of I, no code is run, this is all logic being computed using axioms, inductions, natural deduction etc.

Dafny works like a contrarian, and tries to find 1 example to prove you wrong, compare these two programs

```
method Main() {
  var b;
  assert b;
}
```

the assert tells Dafny that b is a bool

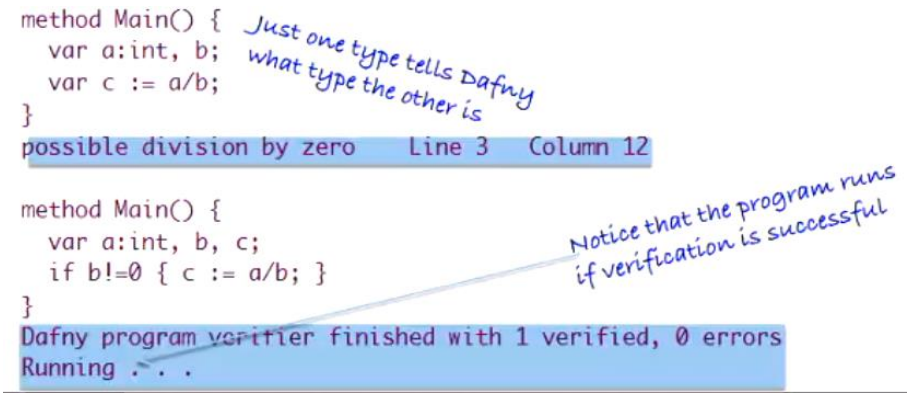
```
method Main() {
  var b: bool;
  print b;
} // no verification in this program
```

here to as be any type

The one on the left, dafny says I know a case where this fails, when b is a Boolean that is equal to false, the assert fails so it will not verify, however the one on the right is almost an identical program but we did not put the leash on our program to say assert and verify, there is nothing to verify. Note Booleans are initialized to false.

However, there are cases where verification happens, even without an assert, this occurs for

1. array bounds
2. division by 0



Dafny spots the potential divide by 0 error before any code is run. In the second example, since the statement is only being run when b is not 0, it is verified.

Verification failure ONLY occurs when dafny can find a counter example to your line of code, in the second example since we took away the ONLY counter example, verification is okay.

Correctness

Dafny can always prove your code is correct, where correct means your code

- guarantee array indices are always within bounds
- contains no null dereference
- contains no division by 0
- guarantee that there can be no run-time errors (very unlikely to pass through Dafny if any runtime errors)
- guarantee the program conforms to specifications
    - if the specification is bad and dafny verifies the program, then the program is just bad.

Dafny doesn't just give you the answer, it requires you to solve the problem declaratively and algorithmically.

- You need to write a specification that says what the algorithm functionally does.
- You need to write a program to then implement that algorithm imperatively.
- You need to do the work, dafny just verifies

What Dafny does

Dafny does 2 things

- Assuming the program terminates
    - Proves the output for the given input
        - Uses a spec of the input and output known as a Hoare triple
            - Give a pre-condition defines what is true before the execution of the line of code
            - Give a post condition defines what is true after the execution of the line of code
            - Line of code between them that executes
    - Proves program terminates

Reality check

Just because Dafny can prove a program is correct in respect to the specification provided, does not mean the program is doing what the customer wants.

It doesn't matter how good the spec and programs are, we always need to black-box test and make sure the program is doing the right thing as REQUIRED by customers.

Verification is all about conformance and making sure it is correct to the specification

Black-box testing is all about confirmation that the code is doing the correct thing required functionally.

We must have both conformance AND confirmation.

Features of Dafny

Dafny contains

- Types
- Methods
- If else statements
- Assignment statements

Preconditions and post conditions

Inbuilt types of Dafny

- `bool` for booleans
- `char` for characters
- `real` for real numbers
- `int` for unbounded integers
- `nat` for natural numbers (the non-negative integers, a subrange of int)
- `set<T>` for an immutable finite, orderless set of values of type T
- `seq<T>` for an immutable finite, ordered sequence of values of type T
  - `string` which is exactly the same as `seq<char>`
- `array<T>` array type of one dimension
- `array2<T>` array type of two dimensions
- `object` is a super-type of all reference types

A ':' is used to indicate a variable has a certain type. For example

```
var a:bool, b:int, c:nat, d:array<int>, e:char;
```

Immutability is crucial for proving correctness, as it makes sure that once it is set, it CANNOT be changed. This avoids all the unwanted variable changes and unexpected behaviors discussed earlier. No re-assigning. Once assigned that's it.

Methods

- Like a procedure in most programming languages
- Input parameters and output arguments which are typed

```
method Abs(x:int) returns(r:int)
{ … }

method AddSub(x:int, y:int) returns(sum:int, dif:int)
{ … }
```

- Input parameters are read-only immutable
- A return statement is not necessary in a method
  - If there is one, it does not need to have any arguments (like return in void function)
    - It will automatically map the return arguments
  - If it does contain the arguments, they should be EXACTLY the ones in the returns statement.
  - We can return multiple things from a method.

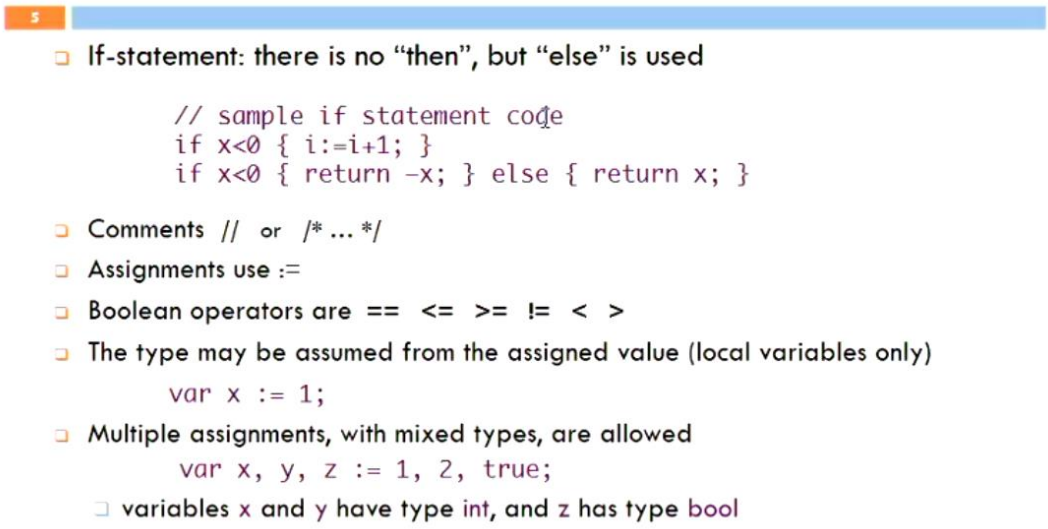
Main method, like all other programming languages.

The verifier in Dafny is run statically at compile-time

- To verify correctness, you do not execute
- You don't need test cases or print statements

- You can use any method name you like, but always start names with uppercase letter (Pascal Case) for methods
- If you do want to execute the program you must include a method with the name Main ()
    - No parameters for main
    - Using this name exactly tells the compiler to produce a .NET executable
    - This method is where execution occurs.

Programming Constructs



Asserts in Dafny

An Assert "assert bool" in Dafny will evaluate the Boolean expression during verification (no code is executed; it is like a compiler which will not run if invalid, C assert will interrupt during runtime)

- Dafny must be able to prove the Boolean is valid (always true)
    - Valid means for all possible execution paths, it is always true (like tautology)
- If the Boolean can ever be false, the verifier reports assertion error.

The aim of the assert is to verify some condition is always true (like invariant)

Dafny has a different language for verification and programming, asserts are in the middle of your program but are only done in compilation, so it is treated differently. Below is some example of more asserts.

```
method MoreAsserts()
{ var a:int := 1;
  var b:int := 2;
  /* 1*/ assert if    a==1    then true  else false;
  /* 2*/ assert if    a!=1    then false else true;
  /* 3*/ assert if    true    then a==1  else false;
  /* 4*/ assert if    a!=1    then a==-1 else a==1;
  /* 5*/ assert if b==(b/2)*2 then true  else false;
  /* 6*/ assert a==-1 ==> b==2;
  /* 7*/ assert b==-1 ==> a==-1;
  /* 8*/ assert a==1 <==> b==-2;

  var c:int, d:int;  // note c and d are declared but not defined!
  /* 9*/ assert c==d || c!=d;
  /*10*/ assert c+d==d+c;
  /*11*/ assert a*(b+c) == a*b + a*c;
  /*12*/ assert c==d ==> true;
  /*13*/ assert c==d ==> false;
  /*14*/ assert if c==d then true else !true;
  /*15*/ assert if c==d then true else true;
  /*16*/ assert if (c==d==>c>=d) then true else !true
}
```

Notice that in verification statements you can use ==> or <==> or you can use if-then-else Ahhh!!!!

8, 13 and 14 are all assertion violations, can be shown with a truth table.

Below is an example showing print in action

```
method Main()
{
  var a, b := '#', '*';
  assert a < b;
  print a, " < ", b
}
```

```
Dafny program verifier finished with 2 verified, 0 errors
Running...
# < *
```

Pre and Post Conditions (specifications)

An ensures statement will declare a post condition. The expression after an ensures must be the predicate (Boolean expression).

Dafny will try to verify the code satisfies the post conditions

```
method Max(a:int, b:int) returns(c:int)
ensures a<=c
ensures b<=c
{
    c := a;
    if a<b {c := b;}
}
```

You wrap the method around some ensure statements, before the code in {} runs. You can also combine the post-conditions into a larger Boolean, ensures a <= c && b <= c

Above, other post-conditions would also verify correctly:

```
ensures a<b ==> c==b
ensures a>=b ==> c==a
```

Keep code clean and simple, don't give too much or too little, try multiple ensures, clean and easy to understand ensures.

Static vs dynamic code

Asserts and print statement happens at different times

- Asserts happen during verification
- A print statement occurs during execution

```
method IsEqual(x:int, y:int) returns (z:bool) // are x and y equal? true/false
ensures if x==y then z==true else z==false
{ z := (x==y); }

method Main()
{
   var v:bool := IsEqual(1,-1);
   if v {print "true";} else {print "false";} // a dynamic check
   assert v;                                  // a static check
}
```

What does Dafny do with this?

Here we have a mix of verification/programming and we need to know what the order of things are. The assert will tell us all we need to know. The program will not compile so no execution occurs. Print statements are only really used for black-box testing. If we take out the assert, then we get rid of the point of dafny and we get the print statement. The assert before run-time will check.

In rubbish ➔ Out Rubbish

If the post-condition (specification) is wrong, then dafny will throw an error

```
method Max(a:int, b:int) returns(c:int)
ensures c==a
{
    c := a;
    if a<b {c := b;}
}
```

```
stdin.dfy(5,13): Error BP5003: A post-condition might not hold on this return path.
stdin.dfy(2,9): Related location: This is the post-condition that might not hold.
Dafny program verifier finished with 0 verified, 1 error
```

- Notice the red and green squiggles in the code
  - The red squiggle is on line 5, position 13 and is where the error occurred
  - The green squiggles are on line 2, starting at position 9 and is the post-condition that was violated

Actually the post-condition is sometimes right (a is the max), sometimes wrong (b is max)!

- the post-condition c==a is satisfiable but it is not valid

Dafny will tell you if there is a case where the post condition is just wrong. In this case, the red squiggle shows WHY the post condition can be wrong, and the green squiggle shows the post condition that was wrong. Remember satisfiable is not the same as valid, IT MUST BE VALID.

Lecture 3

Post Conditions

In all cases, we want a very strong post condition. Take for instance this method Max which will give the maximum of two numbers

```
method Max(a:int, b:int) returns(c:int)
ensures a<=c
ensures b<=c
{
    c := a;
    if a<b {c := b;}
}
```

The post condition above is weak, because it says nothing about C, it just says C could be a or b. we want a strong post condition that says, if a < b, then c is b, and if a >= b then c is a. this is a much stronger post condition as it described the behavior of what the max function is doing!

```
ensures a<b ==> c==b
ensures a>=b ==> c==a
```

In Dafny, we provide our specification as a rule set of what must happen, before and after execution, and everything that lies between that Hoare triple is the code regardless of how or what it does, it

should always meet the specification to compile successful and verify. Any failed assertion will mean, the code will not execute.

Wrong Post-conditions

Here below is an example of the wrong post-condition

```
1  // compute sum and dif of 2 integers
2  method SumDif(x: int, y: int) returns(sum: int, dif: int)
3  ensures dif <= x   // post-condition
4  ensures sum >= x   // post-condition
5  {
6      sum:=x+y;
7      dif:=x-y;
8  }
```

```
stdin.dfy(5,1): Error BP5003: A postcondition might not hold on this return path.
stdin.dfy(3,13): Related location: This is the postcondition that might not hold
    Execution trace:
            (0,0): anon0
Dafny program verifier finished with 1 verified, 1 error
```

The reason it is wrong is that, it will be incorrect if y is a negative number. If y is a negative number, the post condition will not hold! So Dafny picks it up and throws the error instantly!

So how do we fix this?

We can do the following

- Bad option (pre-condition, overloading the function)
    - Adding a pre-condition that y MUST be positive
    - This will limit the portability of our function however, and limit its use
- The better option (stronger post-condition)
    - Diff = x – y
    - Sum = x + y
    - THIS IS THE EXACT BEHAVIOUR WE EXPECT SO THIS IS THE BEST POST CONDITION POSSIBLE FOR THIS FUNCTION SUMDIF

Pre-conditions

We always want the weakest pre-condition (true) but this isn't always possible.

A pre-condition is written in dafny as a **requires** statement like ensures

Example below of SumDif with pre-condition. The precondition will force the program to prove y >= 0 which is called **overloading**

```
method SumDifA(x:int, y:int) returns (sum:int, dif:int)
requires y>=0                // new precondition
ensures dif<=x && sum>=x     // same old post-condition
{
    sum:=x+y;
    dif:=x-y;
}
```

```
Dafny program verifier finished with 2 verified, 0 errors
```

Note however, this is a bad program, since it only works for positive y values, limiting our domain and usability of this function. We don't want to just ignore possible values; we would just be restricting our usability if we do that.

Below are the two different approaches for SumDif

```
method SumDifA(x:int, y:int) returns (sum:int, dif:int)
requires y>=0                // precondition
ensures dif<=x && sum>=x     // weak post-condition
{
   sum:=x+y;
   dif:=x-y;
}
method SumDifB(x:int, y:int) returns (sum:int, dif:int)
ensures dif==x-y && sum==x+y   // strongest possible post-condition
{
   sum:=x+y;
   dif:=x-y;
}
```

Both verify without error, both use the same code.

SumDifA()

- Its pre-condition forces the calling program to prove y>=0
  - This is called overloading
- Its post-condition is too weak (understrength)

SumDifB() is more portable, more robust, better specified

As can be seen, SumDifA is weak, and limits use of the function, a more suitable name would be SumDifPositive

SumDifB however, is more portable robust, and works on any values passed in. Exact same code, different specification. this also helps avoid making functions that don't do what their behavior describes exactly, sure it works for some values, but does it work for EVERY value?

When is it necessary?

Pre-conditions aren't always the devil though, and in some cases are completely necessary. For example, in a binary search, we REQUIRE the iterator to be sorted, otherwise the search functionally is rubbish, we keep halving our bounds for our search on each iteration assuming its ordered, binary search does not work on unordered iterators.

To do this we can say in dafny for our binary search method that

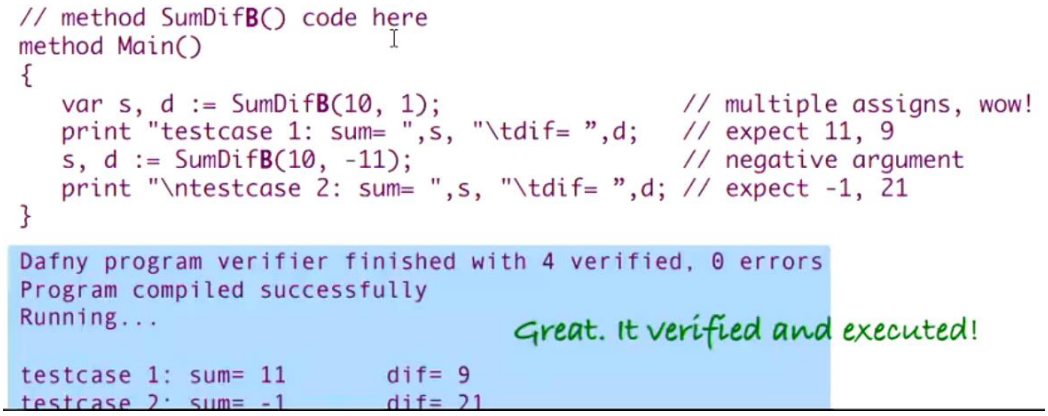
- Pre-condition
    - Iterator is sorted
    - For all pairs in our iterator, the value of the first element, is smaller than the value in the second element
    - For all values in the iterator, the value of the next element is always bigger
- Post-condition
    - The value returned is either
        - The value
        - Not exists

Binary search MUST follow this specification, and how we implement it is up to us.

Testing

How do we test our program gives the correct behavior?

We could try some black-box testing, where we pass in values and see results spat out back at us



But is this the best way to really test the program works? We are saying ourselves that it works but a better way to test this is using asserts.

Take for instance, this function absolute value which will return the absolute value of an integer

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

method TestIt()
{
    var a:= Abs(-7);
    assert a>=0; // expect a positive number
}
Dafny program verifier finished with 2 verified, 0 errors
```

In this case, yes it works, and the assertion ill pass, however this is dangerous as the post-condition is weak, let's say the method just returns the number 5 because it's so random yay, the method to test it will show it still works, as our assert is only checking the post condition holds. A rubbish specification will open doors for dangerous programs.

```
method Abs(x: int) returns(y: int)
ensures y>=0;
{ y := 1; } // s#!t code, but it satisfies the post-condition!

method TestIt()
{ var a := Abs(-7);
  assert a>=0;
}
Dafny program verifier finished with 2 verified, 0 errors
```

- The assert is not checking the result we want
    - Should be, assert a == 7
- The post-condition is not telling us exactly what we want
    - ITS WEAK
- Weak post-conditions and strong pre-conditions will hide and run away from errors

Lecture 5

Slide 26, https://www.cse.unsw.edu.au/~anymeyer/2011/lectures/Week2Dafny.pdf

1
```
method Main() {
  var b: bool;
  assert b;
}
```

2
```
method Main() {
  var b: bool;
  print b;
} // no verification
```

3
```
method Main() {
  var b: bool;
  assert !b;
}
```

What do the above programs do?

What I think is

1. Fails compilation, dafny can find a counter example, if b == false
2. Prints false, since all bools default to false
    a. No verification, so no compilation errors
3. Same as 1, dafny can find a counter example, if b == true
    a. Even if it initialized to false, doesn't mean it works for ALL VALUES!

Harder example

```
method Main() {
  var x:int, y:int;
  print x, y;
        if x<0 { y:=-x; } else { y:=x;}
  assert if x<0 then y==-x else y==x;
  print  if x<0 then y==-x else y==x;
}
```

Which lines are code, which are verification, and what does the program do?

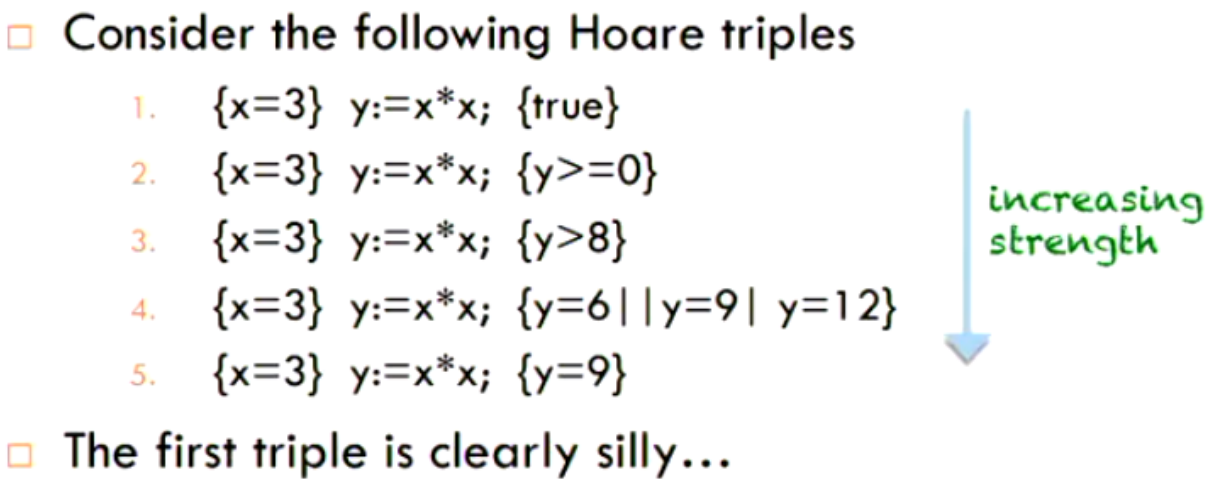
Program, will verify, since the if statement will perform the functionality of our assert. Our print on the other hand occurs after compile time, initializes with 0,

Print 0, 0 and print true for the final print.

Assertions will be checked on the domain before code and prints happen. It will verify the Hoare triple.

Hoare Triples

The weakest condition on anything is, true, it's just saying yes, the state is there

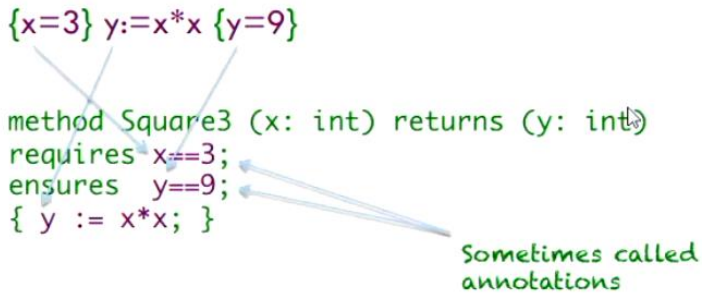


Since the first triple will be equivalent to having no post-condition, but the best possible Hoare triple would be

{x is a numeric type} y:=x*x {y:= x*x} assignment inference.

This is the weakest pre-condition, and strongest post condition.

Below is an example of converting a Hoare triple into a method.



Overloading

- A pre-condition overhead comes from when the method Is called, since Dafny must prove that the pre-condition is in fact true.

It is crucial to use the strongest possible post-condition

- A weaker post-condition is not safe
- A specification defines correctness

It is desirable to use the weakest possible pre-condition

- An overloaded pre-condition can be damaging to the spec and limit the domain
-
- More conditions mean more work

Arrays

- A mutable data structure
- Stored in the heap
- Has the type array<T> where T is a type
- Example: var a: array<bool>
- Length of an array is always defined
    - Range of index from 0 → array.length – 1
- Dafny is an object-oriented language, so we use the new constructor for arrays
- Dafny will always check for array bounds during verification.

```
method Main() {
   var a: array<char> := new char[5];
   a[0], a[1], a[2], a[3], a[4] := 'f', 'r', 'a', 'n', 'k';
   print a[..];                 // whole array
   print a[..a.Length];         // whole array
   print a[0..]; ' '            // whole array
   print a[1..], ' ';           // take off the head
   print a[..a.Length-1];       // take off the tail
}
```

Output:

```
frankfrankfrank rank fran
```

use the first, this can be buggy in verification

```
var a: array := new char['f', 'r', 'a', 'n', 'k'];
```

Short circuiting (protection)

- Dafny evaluates only what it needs to evaluate in order
    - X && Y
        - If X is false, Y will not be evaluated, since it is pointless to check Y after X
    - Order is crucial, like check for null
        - For example
            - 0 <= I <= a.Length && a[i] != 0
            - If the index is out of bounds, we do not want to do the right-hand side of the check first
            - This is like checking for null, then checking your conditions
    - X || Y
        - If X is true then Y is not evaluated, since we know the statement is true
        - Order is crucial just like example above
        - Dafny will exit these full checks early if it already has the answer, dafny is lazy
    - X ➔ Y
        - If X is false, then Y is not evaluated (not X or Y)
        - If X is false, then Y must be true
        - 0 <= I <= a.Length ➔ a[i] != 0, this is how it should be done

Modifications

Since an array is mutable, methods can change values stored in an array

- This is crucial to the verifier
  - If an array changes, every part of the spec that reads the array must be re-verified
- To minimize the verifiers work, we use a modifies clause every time the array is changed

This allows methods to modify mutable data structures

- Below is an example where we DO NOT MODIFY as this causes work for the verifier

```
method TwoMax(a: array<int>) returns (max:int) // max of 2 element array
requires a.Length==2
// a modifies is not necessary here (but causes unnecessary work for the verifier)
{ if a[0]>a[1] {max := a[0];} else {max := a[1];} }
```

- This is an example where it is absolutely needed (no need for tmps super-fast swap)
  - We must include the modify since the array is changed

```
method TwoSort(a: array<int>) // sort a 2 element array
requires a.Length==2
modifies a; // if you forget this here, you will get an error as 'a' is modified
{ if a[0]>a[1] {a[0], a[1] := a[1], a[0];} }
```

Quantifiers

For All

For all is generally used on finite data structures such as arrays

- If a is an array of integers
  - The following expression says none of the elements are -1
    - forall k :: 0 <= k <= a.Length ➔ a[k] != -1
    - this also says if you're outside the bounds you don't care
  - this is very different to this statement, both statements have different truth tables!!!
    - forall k :: 0 <= k <= a.Length && a[k] != -1
    - even if outside the bounds, this will make sure that a[k] != -1, where in the first it says if you're not in the bound, then whatever its true

we usually use implication over conjugation in for all quantifiers as the following can occur

Example: *all girls like diamonds*

- $\forall x$ (girl(x) → likesdiamonds(x))

If we instead use the translation:

- ~~$\forall x$ (girl(x) ∧ likesdiamonds(x))~~
- this says that …
  - … every person is a girl and every person likes diamonds
- This is absolutely not what we wanted to say
- Using conjugation for ∀ is generally wrong.

Exists

The quantifier exists and is like forall, in fact sometimes used with forall.

There is also a membership operator called in, which is a shorthand way of saying there exists.

```
Method ForallExists()
{ var a:array<int> := new int[3];
  a[0], a[1], a[2] := 1, 1, 1;

  assert forall j:: 0<=j<a.Length ==> a[j]==1;
  assert exists j:: 0<=j<a.Length &&  a[j]==1;

  assert !forall j:: 0<=j<a.Length ==> a[j]==2;
  assert !exists j:: 0<=j<a.Length &&  a[j]==2;

  assert forall j:: 0<=j<a.Length ==> a[j]!=2;
  assert exists j:: 0<=j<a.Length &&  a[j]!=2;

  assert 1 in a[..]; // shorthand for exists
  assert 2 !in a[..];
```

Be careful:
if you declare+initialise, Dafny
can 'forget' the assignments
and an exist assertion can fail.

Arrays example

- say we want to initalise an array, where every element, is the index its located at

```
method IndexArray(a: array<int>)
modifies a; // don't forget this: the array is initialised here
ensures forall j::0<=j<a.Length ==> a[j]==j; // this is what we want
{ /* some code in here */ }
```

The code that can go in there could be

- var iterator := 0
- while iterator < a.Length
    - a[iterator] := iterator;
    - iterator++;

```
var i:=0;
while i<a.Length
{ a[i] := i; i:=i+1; }
```

What's the precondition in the method?

There is no pre-condition!

But now we want to prove that the post condition holds, that the result we get is what we want from the code.

We can put in some asserts to assist dafny with this.

We can put in some asserts to 'help' Dafny:

```
{                                  assert true; // the precondition
  var i:=0;                        assert i==0;
  while i<a.Length                 assert ??
  { a[i]:=i;                       assert ??
    i:=i+1;                        assert ??
  }                                assert forall j::0<=j<a.Length ==> a[j]==j; //post-condition
}
```

Having a post-condition means that any loops need an invariant

But we need an invariant for any loop!

Invariants

We can represent a loop as a Hoare triple, like how we represent our methods in dafny.

An invariant I is a predicate expression that says what does not change in a loop

```
...
assert P;      // P is true on entry
while B
invariant I    // notice sandwiched between the 'while' and '{'
{
  S1; S2; …
}
assert Q;      // Q is true on exit
```

I is true on entry, inside, and on exit

- On entry to the loop, P==>I
- Inside the loop, {I && B} S1; S2; … {I}

Our invariant is that on each iteration, I is within the bound

```
method Loop1()
{ var i, limit:=0, 10;
  while i<limit
  invariant 0<=i
  { . . . }
  assert i==limit;
}
Error: assertion violation
```

```
method Loop2()
{ var i, limit:=0, 10;
  while i<limit
  invariant 0<=i<=limit
  { . . . }
  assert i==limit;
}
Dafny … with 1 verified, 0 errors
```

In both methods, the invariant is true in each case, but for method 1, its too weak, whereas in method 2 its very strong. Since method 1 says that I >=0, that is all that can be guaranteed from exiting the loop, and in the case of our Hoare triple, we just get I >= 0 and !i < limit, this doesn't say enough about the value of i.

```
method Loop1()
{ var i, limit:=0, 10;
  while i<limit
  invariant 0<=i
  { i:=i+6;}
  assert i==limit;
}
```

```
Error: assertion violation
```

```
method Loop2()
{ var i, limit:=0, 10;
  while i<limit
  invariant 0<=i<=limit
  { i:=i+6;}
  assert i==limit;
}
```

```
This loop invariant . . . not maintained
Dafny  . . . with 0 verified, 1 error
```

The invariant must be true, before and after the loop. The best invariant which also describes the behavior is the following

```
invariant 0<=i<=a.Length && forall j::0<=j<i  ==> a[j]==j;
```

Here this says that the following is true, before, during and after the loop execution. We make the invariant a weak form of the post condition.

In the method IndexArray() from earlier, we insert the invariant. It consists of a:

- range of the loop index &&
- 'watered-down' version of the post-condition (replace a.Length by 1)

The asserts in the body of the loops are easy to derive from the invariant

```
method IndexArray(a: array<int>)
modifies a; // don't forget this: the array is initialised here
ensures forall j::0<=j<a.Length ==> a[j]==j; // this is what we want
{                          assert true; // the precondition
  var i:=0;                assert i==0;
  while i<a.Length         invariant 0<=i<=a.Length && forall j::0<=j<i  ==> a[j]==j;
  { a[i]:=i;               assert    0<=i<a.Length  && forall j::0<=j<=i ==> a[j]==j;
    i:=i+1;                assert    0<=i<=a.Length && forall j::0<=j<i  ==> a[j]==j;
  }                        assert                  forall j::0<=j<a.Length ==> a[j]==j;
}
```

```
Dafny program verifier finished with 6 verified, 0 errors
```

The post condition should be reflected in the invariant

Note we can also double up the invariant statements like shown below

```
method IndexArray(a: array<int>) // This method just initialises an array ...
modifies a;
ensures forall j::0<=j<a.Length ==> a[j]==j; // ... so this post-condition is easy
{
  var i:=0;
  while i<a.Length                                    ... unlike the next method ...
  invariant 0<=i<=a.Length
  invariant forall j::0<=j<i ==> a[j]==j;
  { a[i]:=i;
    i:=i+1;
  }
}
```

Old mutable values

Arrays are mutable so they may change in a method

- in our post condition or assert, we can specify this exact change to the array
- old(expression) is the value that expression had at the very start of the method before any changes
- comparing exp and old(exp) tells you whether the exp has changed

Method below swaps elements a[p] and a[q] if a[p]>a[q], no other element is affected

```
method PairSort(a: array<char>, p:int, q:int)
modifies a;
requires 0<=p<=q<a.Length;
ensures a[p]<=a[q]; // this is what we want
ensures (a[p]==old(a[p]) && a[q]==old(a[q])) || (a[p]==old(a[q]) && a[q]==old(a[p]));
ensures forall j::0<=j<a.Length && j!=p && j!=q ==> a[j]==old(a[j]);//rest unaffected
{
   if a[p]>a[q] { a[p],a[q]:=a[q],a[p]; }
}
```

After this code has executed, how do you know whether a[p] and a[q] were swapped or not: this is why you need old().

The last ensure says that, if the index is not the ones we swap, the values must have not changed!!! We want to say the behavior of the ENTIRE array if it is used in a method.

Lecture 5

Functions

- only used for specification
- has one return type

```
function IAmOne(): int // yields 1
{ 1 }
```

-
- Fully mathematical, Haskell type functions
- Side-effect free

Function calls cannot be executed

```
method NotMain()
{ assert IAmOne()==1; } // this is good

method Main()
{ print IAmOne();      // WRONG! functions cannot be called in code
                       // should be 'assert IAmOne()'
}
```

Functions only work pre-processing in verification.

See my assignment ex7 for function example very good.

We can combine functions and method to create a function method, but this mixes the verification and executable world, not recommended in my opinion

If a method modifies an array, we need to tell the verifier that we are modifying the array. Similar to this if a function or predicate READS an array, we need to tell the verifier what to read. The function does not allow any executable code, assignments etc. so we must supply to the verifier what we are reading.

- Read statements should not be used in methods
- Modifies statements should not be used in functions
- The argument of reads must be mutable, Reads (mutable)
    - This is because if it's not mutable, we wouldn't have to read anything, we only read dynamic mutable objects.

Predicates

- Used for verification only
- A special function that only returns Boolean
- Used as a true/false checker on judgements
- Needs a reads statement if it accesses an array or object

```
predicate Sorted (a:array<int>)
reads a
{ forall j,k:: 0<=j<k<a.Length ==> a[j]<=a[k] }
```

Notice, no type here, as a function has

(every pair of elements, a[j] > a[k]

This can be used as a pre-condition for a binary search, requires Sorted(a)

Where should we catch errors, in the pre-condition or the predicate?

- Forbidden
    - We forbid certain incorrect inputs

```
predicate forbiddenEqual(a:array<int>)
requires a.Length>=2
reads a
{ a[0]==a[1] }
```

forbiddenEqual() does not allow a.length<2

    - Pre-condition says any array less than length 2, go away not going to do anything uh uh
- Forgiven
    - We will allow incorrect inputs and return true because reduction from absurd

```
predicate forgivenEqual(a:array<int>)
reads a
{ (a.Length>=2) ==> a[0]==a[1] }
```

forgivenEqual() returns true if a.length<2

    - This can be dangerous since we can say an array of length 1 has equal values. Wrong behavior returning true???

- unforgiven
    - same as forgiven, but will return false if the array is bad

```
predicate unforgivenEqual(a:array<int>)
reads a
{ if (a.Length>=2) then (a[0]==a[1]) else false }
```

Now comes the question, which one do we use?

- On good data, these all return the same result
- These predicates only behave differently on bad data
- We need to decide what to do with bad data, this is our role as project managers
- Usually forbidden is best since it blocks bad data giving undefined behavior, rubbish in == not accepted
- But sometimes we want forgiven/unforgiven

Example

Writing a method called find

- Returns the first index of an array with a specified key value
- Post conditions
    - If the key is found, I > 0, then that implies that its within array bounds and a[i] does contain key
        - I >= 0 ➔ (I < a.Length && a[i] == key)
        - I >= 0 && I < a.Length ➔ a[i] == key, THIS IS WRONG
            - If I > a.Length it returns true, since false ➔ X is always true
    - If the key is not found, then that implies the key does not exist within the array
        - I < 0 ➔ (forall k :: 0 <= k < a.Length ➔ a[k] != key)
    - Our specification

Requirements

No other part of the work so cripples the resulting system if done wrong, no other part is more difficult to rectify later (Brooks 1987) ➔ requirements

"I'll go talk to the stakeholders and find out their requirements... in the meantime, you guys start coding."

KEKW ET TU

- The most important phase in software development is requirements
    - Requirement elicitations (validation)
        - Make sure the product is what the client expects
        - Most important! If the product is wrong, the code will be wrong (even if the code is correct, if it's not doing what it's supposed to do, it's wrong)
    - Verification and testing
        - Make sure the code behaves to satisfy the requirements

Requirements should be structured

- They shouldn't be a shopping lost
    - "oh, I have an idea, just add it to the list"
- They should be hierarchical and contain structure
    - Top level root → business/stakeholders/executive requirements
        - Not too technical, used to communicate with business owners and someone who kind of understands the domain, no jargon
        - Taken from perspective of the business not the programmers
            - At Maia, even though I wanted to change a feature to look better, it wasn't a part of the requirement
        - Define what needs to be built, blackbox definition don't care how
        - Can be seen as mini goals that build to the company products
    - Programming tasks
        - Leaves, programming tasks that say how its implemented

Types of requirements

- Functional requirements
    - Specify behavior
        - "system x must perform y to achieve z"
- Nonfunctional requirements, they describe how the product should turn out to be
    - High performance
    - Reliability
    - Maintainability
    - Security
    - Difficult to measure
    - Can't be put into a use case
    - How does the customer know they are satisfied?
- Good design allows you to modify components very easily without changing the behavior, a badly structured project will lead to undesirable and unmaintainable results

Priorities (Triage)

Adding priorities at the beginning is very important and gives our requirements structure

- Priorities allow us to think in respect to the time scope
    - What is core business and must be implemented? What must we show?
    - What can wait for future versions
- It means you think about layering and scoping
    - Extend the scope (add layers) later by implementing low priority requirement systems
- All extra features and not core requirements can be released in future versions
- We need to decide which requirements are the critical and core requirements, and which are extra features that can wait
    - A trade-off due to limited funds, time and resources
        - For each requirement, a cost to benefit analysis and dependency analysis should be informally done
        - Requirements that are costly, time consuming or resource hungry are placed at a lower priority
        - Sort of how I manage my time for university/work
- It is very important that the clients who are asking for the system to be built agree to the prioritization of the requirements.

MoSCoW requirements

- Must have requirements – priority 1
    - Critical, without these the project will fail
- Should have Requirements – priority 2
    - Important, implement if there is time or wait to a new version if pushed to release
- Could have requirements – priority 3
    - Desirable, worth doing if there is time
- Won't have (but would be nice) – priority 4
    - Unlikely to be included but may be in future releases
    - A path for the future of our system

Priority consistency and criticality

- As we go down the tree of requirements, the priority should decrease
    - The root should be business requirements
- Get requirements from the client to determine priority
    - This allows us to determine critical requirements
- We must limit the critical requirements to the core that can be built in the time available, and necessary to run
    - If this is not the case the project is doomed from the start
    -

- Ancestral dependence
  - The priority of a child requirement must be lower than the parent

    Req1: set username and initial password [Priority 3]
    Req1.1: change password [Priority 3]
    Req1.2: handle forgot-password [Priority 2] ?
  -
- Cross dependence
  - If requirement A depends on requirement B, then requirement A cannot be lower priority than requirement B.

    Req3: create account for user [Priority 2]
    Req1.1: able to 'edit profile' [Priority 2]
    Req1.2: show 'billing history' [Priority 2]
    ...
    Req6: Billing [Priority 1]
    Req6.1: add invoice to account 'billing history' [Priority 1]
  -
  - In this example, requirement 6 cannot have a higher priority than requirement 3, since we need requirement 3 for 6
- Identify stakeholders

- Priority order
  - Most natural order to present the reqts.
    - In this order, the stakeholders are referred to explicitly for each reqt.

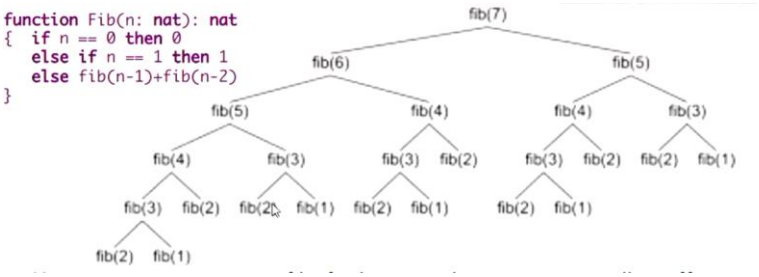
- Stakeholder order
  - If the system involves stakeholders that have quite different functionality, then presenting the requirements of each of the stakeholders separately can be clearer.
  - This order comes at a cost: it means that there are multiple lists of priorities at each level, making dependencies and overlap difficult to recognise.
    - A substantial overlap in functionality makes this a poor choice as there is a high probability that the same requirement for different stakeholders will be implemented differently and separately, leading to inefficiencies and inconsistencies.
- A mixture of orderings, with the major ordering being priority is also possible.

Lecture 6

Recursion

Recursion is a mathematical algorithmic tool that allows us to recursively call our function.

A classic example Fibonacci numbers



- Using recursion to compute fibs for large numbers is exponentially inefficient and can lead to STACK OVERFLOW errors
    - Colossal
    - Fib(40) has about 1 trillion nodes in the tree
    - Fib(80) has about 1 trillion trillion nodes which would take about 100 million years to do

Computationally speaking, it's not good for getting what we want, but rather it is good for showing what we got is good

To get this right

- Verification ➔ recursive since we can mathematically talk about this
- Execution ➔ iteratively (much cleaner and faster)

Fast, linear Fibonacci computation

```
method IterativeFib(n: nat) returns(b: nat)
{
   if (n==0) {return 0;}
   var a := 0;
       b := 1;
   var i:int := 1;
   while i<n
   {
      a, b := b, a+b;   // b == Fib(i+1)
      i := i+1;         // next i
   }                    // i==n after the loop
}
```

Two worlds, we know what we want is the recursive definition, and how we do it should be the iterative definition for speed, we use the recursive definition as our specification.

- To compute Fibonacci numbers:
    - obviously must use the iterative Fibonacci
- To verify the iterative definition:
    - use the recursive Fibonacci function
        - it defines the behaviour perfectly
        - it is used as a property definition
        - the colossal tree is not evaluated of course
- In essence:
    - Annotate the iterative algorithm with
        - A post-condition that calls recursive Fibonacci
        - An invariant that also calls recursive Fibonacci

Here is a beautiful example of combining specification, and code. The mathematical definition, and the implementation.

```
function Fib(n: nat): nat
{
   if n==0 then 0
   else if n==1 then 1
   else fib(n-1)+fib(n-2)
}

method IterativeFib(n: nat) returns(b: nat) // this is real code
ensures b==fib(n)
{
   if n==0 {return 0;}
   var a := 0;
       b := 1;
   var i:int := 1;
   while i < n
   invariant 0<i<=n && a==fib(i-1) && b==fib(i)
   {
      a, b := b, a+b;
      i := i+1;
   }
}
```

Purple = compute, green = verify

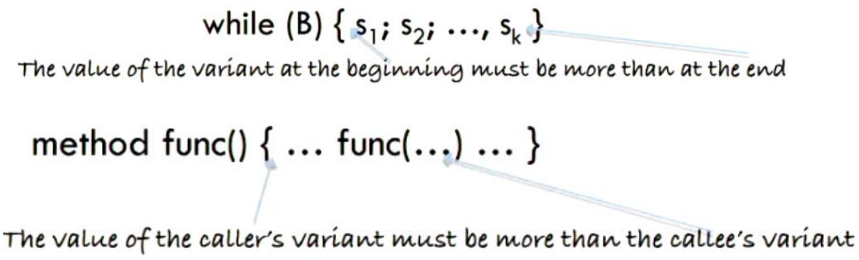
SAME I DID FOR MY ASSIGNMENT IN Q7!!!

The invariant is where the specification and code sort of marry and will DESCRIBE the code. It is the way that dafny knows what happens AFTER a while loop.

Termination

- Dafny can prove a program will terminate
    - We need this to prove total correctness (Refer to earlier slides)
- Failure to terminate comes from
    - Loops
    - Recursive calls
- To prove termination Dafny needs a
    - Variant
        - Defines a value that changes on each repeated call
        - Usually a value that decreases to 0, example counter decreasing
- The variant lets Dafny Prove
    - Progress in a loop/recursive call
    - Boundedness

The variant in loops



In dafny we can give our variant with the keyword Decreases

- The decreases expression must decrease on each evaluation
- The expression must be bounded (default 0)

```
// iterate down
while (0<i)
invariant 0<=i;
decreases i
{
   . . .
   i:=i-1;
}
```

```
// iterate up
while (i<n)
invariant 0<=i<=n
decreases n-i
{
   . . .
   i:=i+1;
}
```

```
// recurse around
function func(i: int): int

decreases i
{
   . . .
   func(i-1);
}
```

- Dafny (on cse machine) is smart enough to work this out

The `decreases` statement helps the verifier prove `Mul()` terminates

- `x` starts positive, and eventually `x` will equal 0

```
method MulByAdd(x: int, y: int) returns (r: int)
requires x>=0 && y>=0
ensures r == x*y
decreases x  // actually not necessary as Dafny sees this is what happens
{
   if x == 0 {r := 0;}
   else {var sum := MulByAdd(x-1, y);  r := sum + y;} // x is a counter
}
```

Sometimes however it is necessary to provide a decreases statement take for example this program with a tricky loop condition

Dafny cannot prove that the loop in the following program terminates.

```
method Main()
{ var i, j := 0, -1;
  var counter, limit := 0, 100;
  while (i!=j) // a tricky loop condition: is i the loop index?
  invariant 0 <= counter <= limit
  invariant counter==limit ==> i==j;
  // decreases limit-counter
  {  j := i;
     counter := counter+1;
     if (counter < limit) { i := i+1; }
} }
```

The above program results in:

```
stdin.dfy(4,5): Error: cannot prove termination; try supplying a decreases clause
```

Note this code is rubbish, what the hell who wrote this trash, dafny cannot infer an invariant from this rubbish code.

Dafny can be smart and tell you if your code won't terminate

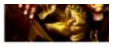
```
method Loop1()
{
   var i := -1;
   while (i!=0) {
     i := i-1;
   }
}
```

```
method Loop2()
{
   var i := 1;
   while (i!=0) {
     i := i+1;
   }
}
```

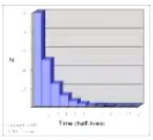
Error: cannot prove termination; try supplying a decreases clause for the loop

Another example of termination

Dafny reports a termination error for the following method



```
method Decay() {
   var x: real := 2.0;
   while (x!=0.0)
   {
      x := x/2.0;
   }
}
```



Error: cannot prove termination; try supplying a decreases clause for the loop

- Logically, the variant x always makes progress

Even though we decrease to 0, we want to show it will be 0 eventually, but it cannot be reached.

Even adding a decreases x clause will not help, since it will say it might not decrease to the bound.

We can hackly fix this

```
method BoundedDecay()
{
   var x := 2.0;
   var i := 10;
   while (x!=0.0 && i>=0)
   decreases i // must include this
   {
      x := x/2.0;
      i := i-1;
   }
}
```

The decreases clause is necessary because Dafny doesn't know what this expression will do

But we must supply the decreases clause since it's not smart enough to know the conjunction of the two is dependent on each side of the expression. In fact, Dafny won't even compute finite precision it will symbolically expand the logical equivalent

Example

```
method BoundedDecay()
{
   var x := 2.0;
   var i := 10;
   while (x!=0.0 && i>=0)
   decreases i
   {
      x := x/2.0;
      print x;
      i := i-1;
   }
}
method Main()
{
  BoundedDecay();
}
```

```
Dafny verifier finished with 2 verified, 0 errors
Program compiled successfully
Running  ...   (2.0 / 2.0)
               (2.0 / 4.0)
               (2.0 / 8.0)
               (2.0 / 16.0)
               (2.0 / 32.0)
               (2.0 / 64.0)
               (2.0 / 128.0)
               (2.0 / 256.0)
               (2.0 / 512.0)
               (2.0 / 1024.0)
               (2.0 / 2048.0)
```

Dafny floating point works symbolically. The verifier doesn't actually ever evaluate x

(If you want to exhaust the precision of the computer, then this solution is unsatisfactory)

Assignment 1/Project house keeping

- Don't program a solution
- Don't write algorithms unless
    - To prove a set of steps like a post condition

Write the post condition and have as little code as possible.

Write the SMALLEST possible program to get you where you want then to go about proving it

PROJECT NOTES

- Use cases
    - https://www.usability.gov/how-to-and-tools/methods/use-cases.html
- Spread work
    - Delegate responsibilities
- Use cases and requirements MUST BE STRONGLY LINKED
    - We need to define our problems and how we perceive it
        - Add pre + post conditions to use cases (is good anyways)

Dafny is an abstraction from reality, its logical. We can artificially create these real-world examples (create a dafny program to drink water)

Specification is not specific on purpose

- We can define the system for vampire, we create and define it
- don't write a specification of the WHOLE system, we need to implement something
    - create MOSCOW requirements for a subsystem, for a small group of stakeholders
    - can create stakeholders for certain functionality e.g. a blood dump entity for disposing of old blood

we can take part of the system such as the hospital that is rich, we want to prove algorithms, we want algorithmic stuff and code we can verify correct

the richer and more verifiable algorithms are ➔ higher mark

don't just create a simple database system that just updates numbers and check if they are correct, create algorithms, we want to sort, write up VampyrSort, we want to search? Create VampyrSearch as long as there are algorithms, don't just throw the work to higher level systems and have no algorithms to verify.

- Have a few GOOD requirements

code that can be correct includes

- sorting algorithm
- search algorithms

Lecture 7

Work breakdown Structure

A Work Breakdown Structure is a hierarchical decomposition of the work to be executed by the team in order to

- accomplish the project objectives
- create required deliverables

the most important thing is that a wbs is outcome driven not task oriented,

- not about how we do things
- rather about the outcomes we expect
- generating deliverables to make rather than the nitty gritty

100% rule

- collectively: the sum of the components equals the whole
- exclusivity: no overlap between any components

WBS is all about the what rather than the how

No verbs no verbs! Only nouns and adjectives

Child elements of a WBS have 100% of the scope their parents, for example

- component has 50%
    - break down component into 10 components
    - all 10 components must cover that 50%
        - split however we like

Purpose of WBS

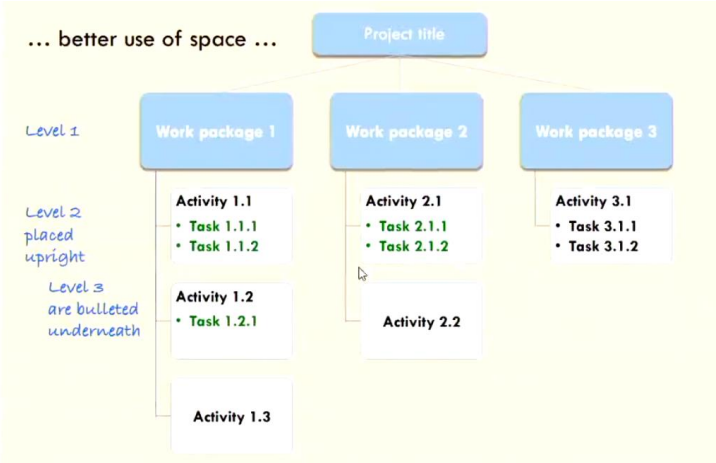
- identifies the “work packages”
    - something that has an outcome
- shows all the deliverables
- tracks costs/progression and performance
    - resource management
- Define responsibilities
- Identifies where coordination is required

Example WBS (note activity should not be there really, what to be done, not how they are done)

Give a team the requirements and what needs to be done, let the team be dynamic, flexible and manage their own time and tackle it their way through scheduling

Lowest level of the tree can be the verb, but only at the lowest level

Below is an example of a more compact WBS



Create a dictionary and glossary for WBS

- Make it known what terminology means to any users

Work packages of a WBS

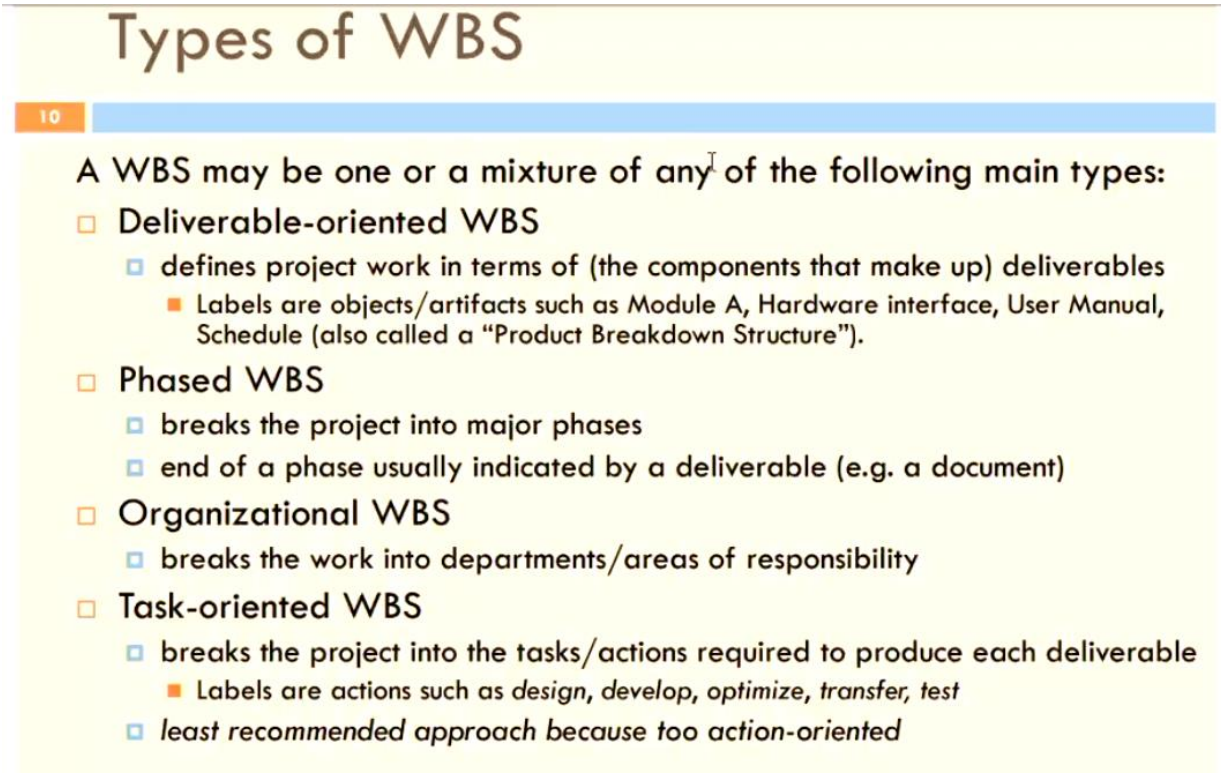
The first level is the most important level, it usually determines

- Costs
- Scheduling

Deliverables are the output of the packages

Every WBS element consists of an element that has

- Scope of work, deliverable
- Budget
- Responsible person's name
- Maybe a time estimate
- A goal

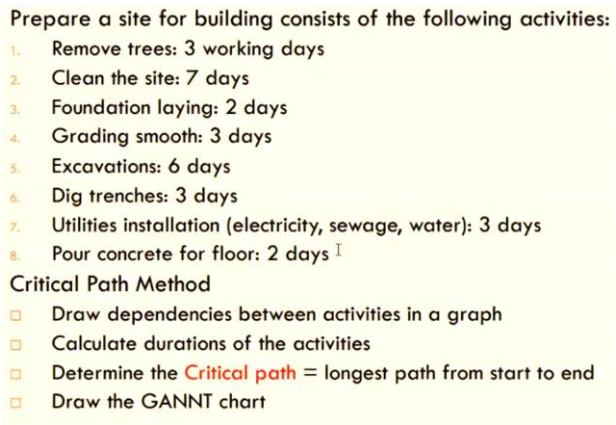


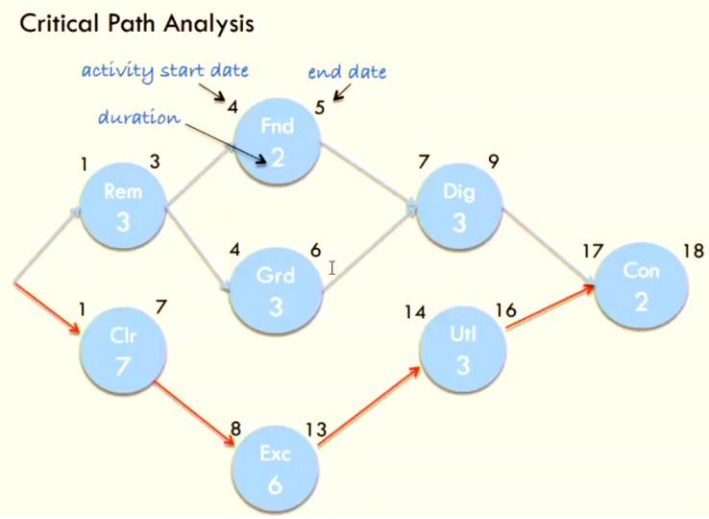
A WBS is not a schedule, it is a specification to follow that outlines the scope of the project

Critical Path Method (CPM)

- The sequence of scheduled activities that determines the duration of the project
- It is the longest sequence of said tasks to meet the project deadline
- Any delay on the path means the entire project is delayed
- Most projects only have one said path

CPM example





Red path is the CPM, and this can be represented in a Gantt chart

WBS Frame

Note in the report make major deliverable include

- Front end
- Back end
- Final report
- Project management

Structure from there

Dafny

Sets

A set is like an array

- Odorless
- Elements are immutable, once created it cannot change, cannot write to sets
- Each element appears only just once
- Mathematical set

- so the sets {1,1}, {1,1,1} etc are all equivalent to {1}
- the set {1,1,2,3,3,3,4} is equivalent to {1,2,3,4}

Syntax

```
set<T>    where T is a type such as int, bool etc
```

Usage

- Examples

```
var s1:set<int> := {};          // the empty set
var s2:set<int> := {1, 2, 3};   // set contains 1, 2 and 3
var s3:set<int>, s4:set<int> := {1,2}, {1,4};
```

- Arithmetic on sets:
  - Compare `s3 != s4` `{1,1,2,3,3} == s2`
  - Union `s2 + s4` `s3 + {3}`
  - Intersection `s2 * s3` `{2,3} * s4`
  - Difference `s2 - s3` `{1,2,3,4} - s2`
- Can be used in annotations
  - `assert s3 * s4 == {1};`

- Program showing type declarations are not necessary.

```
method CheckSet()
{
  var countup := {1,2,3,4,5,6,7,8,9}; // notice no type declar.
  var countdown := {9,8,7,6,5,4,3,2,1};
  assert countup - countdown == {};
}
```

- Sets are very flexible.

```
assert {1} <= {1,2} && {1,2} <= {1,2};          // subset
assert {} < {1,2} && !({1} < {1});              // proper subset
assert !({1,2} <= {1,4}) && !({1,4} <= {1,2}); // no relation
assert 5 in {1,3,4,5};                          // membership
assert 2 !in {1,3,4,5};
```

- You can test for size using vertical bars, e.g. `|s|`

Sequences in Dafny

- Like sets but ordered
- Models stacks/queues/lists

```
method SeqAreOrdered()
{
    var s: seq<int> := [2,1,3]; // arrays use 'new', seqs do not
    var t: seq<int> := [1,2,3]; // but both use [ ]s
    assert s==t;
}
```

```
stdin.dfy(9,10): Error: assertion violation
...
Dafny program verifier finished with 2 verified, 1 error
```

Note, this program verifies if s and t are sets.

```
method Main()
{ var a:array<int> := new int[][1,2,3,4]; // mutable
  var s:seq<int>   := [1,2,3,4];          // immutable
  var t:set<int>   := {1,2,3,4};          // immutable
  print a.Length, ' ', |s|, ' ', |t|;
}
```

Since sets and sequences are immutable, they don't need a reads statement.

Slicing Iterators

We can use subsequences

- The indices of an array of length n are 0 → n – 1
- **A subsequence s[i..j] is a sequence consistent of elements from I to j – 1**
- A subsequence is known as a slice

```
1 method SlicingTricks()
2 {
3    var s := [1,2,3,4,5];                // notice no type
4    assert s[0]          == 1;           // head element
5    assert s[0..1]       == [1];         // head slice
6    assert s[|s|-1]      == 5;           // tail element
7    assert s[|s|-1..|s|] == [5];         // tail slice
8    assert s[..|s|-1]    == [1, 2, 3, 4]; // slice with no tail
9    assert s[1..|s|]     == [2, 3, 4, 5]; // slice with no head
10   assert s == s[..] == s[0..] == s[..|s|] == s[0..|s|]; // whole slice
11   assert s[|s|..|s|]   == [];          // empty
12 }
```

- The head element is `s[0]`, and head slice `s[0..1]` (lines 4 and 5)
- The tail element is `s[|s|-1]`, and tail slice `s[|s|-1..|s|]` (lines 6 and 7)

- **Unions**

```
assert [1] + [2,3,4] == [1,2,3,4];
assert [] + [1,2,3,4] == [1,2,3,4];
```

- **Universal quantification**

```
assert forall i :: 0<=i<=|s| ==> s==s[..i]+s[i..];
```

e.g. if i==3 then s==[..3]+[3..] ==[1,2,3]+[4].
This assert says that you can do this for all indices

- **Membership**

```
var s := [1,2,3,4,5];
assert 5 in s;  // this is true
assert 0 !in s; // this is also true
```

```
method RevPrint(s: string) returns()
{
  if s==[] {return;}          // end of string, finished recursion
  else {
    print s[|s|-1];           // print the tail character
    RevPrint(s[0..|s|-1]);    // recurse on the front slice
    return;
  }
}
```

In operator

The in-operator allows us to check membership and do cool checks, it works on sets and sequences

This however does not work in arrays, only in sets, multisets or a sequence

Example of using a set/sequence for updating values

Replace the element at index i of sequence s[] by value v

```
function patch(s:seq<int>, i:int, v:int): seq<int>  returns a new seq
requires 0<=i<|s|
ensures patch(s,i,v) == s[i := v]   update operation: seq s with s[i]:=v
{
   s[..i] + [v] + s[i+1..] // replace element at s[i]
}  slice to index i-1  + new element + slice from i+1 onwards
```

Now call the function and test

```
method AssertPatch()
{  var s:=[10,20,30,-1,50];                    // a 'bad' sequence
   assert patch(s, 3, 40) == [10,20,30,40,50];  // patch it up
}
```

We don't update values; we just create a new set/sequence with the correct data

Converting a sequence to an array and vice versa

```
method Main()
{
  var a: array<char> := new char['c', 'a', 't'];
  var s: string; // synonym for seq<char>
  s := a[..];  // copy array into seq
  print s;
}
```

```
Dafny program verifier finished with 2 verified, 0 errors
Running ...
cat
```

Obvious 'Conjecture'

19

Consider the 'conjecture':

It is impossible to find positive integers a, b, and c that satisfy the equation $a^n + b^n + c^n \leq 0$ *for integer* $n \geq 0$

Obvious: if a, b, and c are positive, the sum of the terms can never be zero or negative.

Here is the 'conjecture' in Dafny.

```
method Obvious3(a:int, b:int, c:int) returns (possible:bool) // n==3 case
requires a>0 && b>0 && c>0;
ensures !possible;
{if a*a*a + b*b*b + c*c*c <= 0 {possible := true;} else {possible := false;}}
```

```
Dafny program verifier finished with 1 verified, 0 errors
```

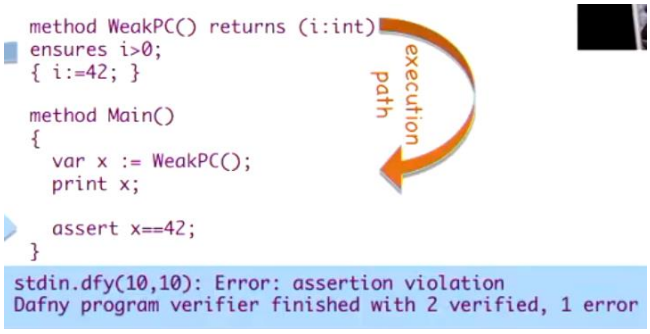
The method is verified, so Dafny has proved there is no a, b, c

Fermat's last theorem

It is impossible to find positive integers a, b, and c that satisfy the equation $a^n + b^n = c^n$ *for integer* $n>2$. *Pierre de Fermat 1637*

Lecture 8

Sneak assertions

- Just because your code prints the right result doesn't mean you have proven its working as intended
- Your post condition is the only thing you can assert
- THE ONLY THING YOU CAN ASSERT FOR YOUR METHOD IS ITS POST CONDITION
    - Example
        - 

- A bad specification can never be proven. In the example above if we say the method will just return a number bigger than 0, we cannot know anything about the number other than the fact its bigger than 0
    - The weak post condition needs to be changed to say that ensures I == 42
    - We can also modify the assert, so we can say assert x > 0, but the first option is more desirable

We always want the strongest post condition; we only assert the strongest post condition.

Assertion witnesses

- Compare these two dafny programs
    - ```
      method Obvious3(a:int, b:int, c:int) returns (possible:bool) // n==3 case
      requires a>0 && b>0 && c>0;
      ensures !possible;
      {if a*a*a + b*b*b + c*c*c <= 0 {possible := true;} else {possible := false;}}
      ```
    - ```
      method Fermat3(a:int, b:int, c:int) returns (possible:bool)
      requires a>0 && b>0 && c>0;
      ensures !possible;              // this is Fermat's Conjecture
      {
        if a*a*a + b*b*b == c*c*c {possible := true;} else {possible := false;}
      }
      ```
- The first will verify easy
- The second will not

Dafny can easily prove atomically the first program, as obvious as it looks

The second program is something dafny CANNOT prove, dafny isn't a mathematician, it is your yes man.

This also occurs in some mathematical theorems

See this example

```
predicate Pythag(a:int, b:int, c:int)
requires a>0 && b>0 && c>0
{ a*a + b*b == c*c }

method VerifyPy() {
  assert exists l,m,n::l>0 && m>0 && n>0 && Pythag(l,m,n);
}
```

If we try to assert there exists an a, b, c that satisfies PYTHAGOROS

- WRONG ERROR ASSERTION violation
- Even though we know 3, 4, 5 will satisfy that predicate, it does not check each value

Dafny doesn't prove theorems it checks theorems

See this example too

```
method bePositive() {
   assert exists x :: x > 0;
}
```

So obvious, there are 0 → infinite cases where it exists, but dafny doesn't do the logic work for you

```
assert exists x :: x>0 && x*x<25 && x*x*x>27;
```

Here again, answer is 4, only answer

- Assertion violation

We can't ask dafny to prove stuff for us, we need to prove it dafny will verify

De-triggering an exists quantifier

Provide a predicate/function that contains the expression with quantifier

```
predicate is3(a:int) { a>0 && a<=3 && a%3==0 } // solution a = 3
predicate pos(a:int) { a>0 }                   // solution a > 0

method bePositive()
{
 assert exists x :: x>0;                  // trigger warning, assertion violation
 assert exists x :: pos(x);               // assertion violation

 assert exists x :: x>0 && x<=3 && x%3==0;  // trigger warning, assertion violation
 assert exists x:: is3(x);                // assertion violation
}
```

We turn our asserts into a predicate, this will fix our trigger error, since now we define what x is in our predicate, dafny has its trigger, but this won't fix the assertion violation

- Remember DAFNY DOESN'T PROVE IT VERIFIES

To do this, we need to give a case where it exists, then we can assert exists

We give a witness for exists quantifiers to fix this

See example below

Provide an instance to 'help' Dafny prove existence

```
predicate is3(a:int) { a>0 && a<=3 && a%3==0 } // solution a = 3
predicate pos(a:int) { a>0 }                   // solution a > 0

method bePositive()
{
  assert pos(1234);                              // instance predicate
  assert exists x :: pos(x);
//assert exists x,y :: pos(x) && pos(y) && x!=y; // assertion violation
  assert pos(5678);                              // instance predicate
  assert exists x,y :: pos(x) && pos(y) && x!=y;

  assert is3(3);
  assert exists x:: is3(x);
  assert !is3(1);
  assert exists x:: !is3(x);
}
Dafny program verifier finished with 2 verified, 0 errors
```

We can also use sets to limit our domain (range of values) thus allowing dafny to prove with a set range of values

```
predicate Pythag(a:int, b:int, c:int)
requires a>0 && b>0 && c>0
{ a*a + b*b == c*c }
method PythagInSet() {
   var s: set := {9,8,7,6,5,4,3,2,1};
   assert exists     x::x in s;
   assert exists     x::x in s && x<4;
   assert exists   x,y::x in s && y in s && x==y+1;
   assert exists x,y,z::x in s && y in s && z in s && Pythag(x,y,z);

   var t := s - {5};              // '5' has been removed from set 's'
   assert !exists x,y,z::x in t && y in t && z in t && Pythag(x,y,z);
}
Dafny program verifier finished with 1 verified, 0 errors
```

Since we use a set, we don't actually need to provide a witness, since dafny can TRY with your limited domain. Note it won't tell you the solution it will just tell you there is a solution.

We should always put expressions with quantifiers in a predicate

```
predicate Add1(x:int) { x+1>x }
method forallAssert() {
  assert forall x:: x+1>x;   // Warning: /!\ No terms found to trigger on.
  assert forall x:: Add1(x); // Okay
}
```

Not having a trigger is under-specification, we aren't proving random facts we usually provide triggers when we check something.

Ghosts

We could also provide ghosts to get around this

```
method forallBetter() {
   ghost var x:int;  // the compiler will never see this variable: it is a ghost
   assert x+1>x;     // NO QUANTIFIER NECESSARY!
}
Dafny program verifier finished with 1 verified, 0 errors
```

The ghosts are only used in verification, and only exists in the verification world, we can't print with it or execute it, we prove with ghosts

Completeness and Soundness in Dafny

- We can get false negatives
- False positives
- True negatives
- True positives

- In logic, the ideal model is:
  - Complete: there are no false positives, but there may be false negatives
    - all errors it finds are real, but it may miss some errors
      - hence it may accept 'bad' programs (...gasp, horror, this is unsafe)
  - Sound: there are no false negatives, but there may be false positives
    - It does not miss errors, but some errors it finds may be spurious
      - hence it may reject 'good' programs (...unfortunate, but safe)

We can't have both sound and complete, we can only "choose one"

In dafny we want it to be sound, we want to make sure there is no false positives, that is the core of dafny. However, in saying this there may be false negatives, where a program is correct but dafny says it can't prove it, this is the trade-off. Dafny is spurious

- Dafny will report an error if …
  - it doesn't have the rules/knowledge (quite common)
  - it has reached a time limit (possible)
  - it has run out of storage (rare)
- Dafny terminates with no message (… always a timelimit problem??)
- Dafny reports "oops …encountered an issue" (a problem in the verifier, rare)
- 'rise4fun' reports the server is too busy (rare)

Dafny says "no errors", then there really are no errors, but …

- **A weak/wrong spec can hide a buggy or even wrong program**
- **Extreme case (ensures true) means any program that compiles is correct**

we can force a method to verify by ensuring true or weak post conditions or awfully strong preconditions. This doesn't mean its correct, remember rubbish in == rubbish out.

All correctness comes from our methods post conditions!

Model-Driven Engineering

- Start with a model and a set of concepts broken down into implementation
  - Sometimes It means a graphical-based development method with automatic code generation
- The model forms the basis of the design
  - Lowest level is implementation
- Design it, hit the button, generates the code to match the design specification
- The flaw however is that not everything can be implemented, we sometimes want to implement something unimplementable example
  - A program that says 3 == 4 is true,
- The model drives the development, it explains what we want, and the code is what's in the middle similar to Hoare logic
  - {spec1} code {spec2}
- In dafny we will fix a post condition and invariants, then we write the code to fill it in, making sure our code matches our specification.
  - You don't just change requirements when you want to do it a certain way

Example cube number ONLY using addition

Start with desired result "C program"

```
// compute the cubes of integers 0..num
// returns num³, without verification
method Cube(num:int) returns (cube:int)
{
  cube:=0;
  var n:=0;
  while n<num
  {
    n:=n+1;
    cube :=n*n*n;
  }
}
```

Now we have to add an invariant and post condition this is step 1

```
// now add verification
method Cube(num:int) returns (cube:int)
requires num>=0;
ensures cube==num*num*num; // love this!
{
  cube:=0;
  var n:=0;
  while n<num
  invariant n<=num;
  invariant cube==n*n*n;
  {
    n:=n+1;
    cube := n*n*n;
  }
}
```

Note how our cube is occurring AFTER we update n, so actually in the loop iteration

Cube = (n+1)^3

So, we can put the cube before the update, and say that cube == (n+1)^3, expanding that out we get the following

```
method Cube(num:int) returns (cube:int)
requires num>=0;
ensures cube==num*num*num;
{
  cube:=0;
  var n:=0;
  var k:=1;
  while n<num
  invariant n<=num;
  invariant cube == n*n*n;
  invariant k == 3*n*n + 3*n +1;
  {
    cube := cube + k;
    n := n + 1;
    k := 3*n*n + 3*n +1;
  }
}
```

```
Dafny program verifier finished with 1 verified, 0 errors
```

Note that

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

And the cube identity with (n+1)^3 = n^3 + (3n^2 + 3n + 1) ← k

Now we have gone from cubed to square, Break down even further

now note that k is using the value of n + 1

so, k = 3(n+1)^2 + 3(n+1) + 1

expanding it out we get

k = 3(n^2+2n+1) + 3n+3 + 1

k = 3n^2 + 6n + 3 + 3n + 4

k = 3n^2 + 3n + 1 + (6n+6)

k = k + (6n + 6)

so, we can re-express this as the following

```
method Cube(num:int) returns (cube:int)
requires num>=0;
ensures cube==num*num*num;
{
  cube:=0;
  var n:=0;
  var k:=1;
  var m:=6;
  while n<num
  invariant n<=num;
  invariant cube == n*n*n;
  invariant k == 3*n*n + 3*n +1;
  invariant m == 6*n + 6;
  {
    cube := cube + k;
    k := k + m;
    n := n + 1;
    m := 6*n + 6;
  }
}
```

Finally, we can break it down even further to remove the multiplication, since n = n + 1, we get

M = 6(n+1) + 6

M = 6n + 6 + 6

M = 6n + 6 + (6)

So, we can finally say

```
method Cube(num:int) returns (cube:int)
requires num>=0;
ensures cube==num*num*num;
{
  cube:=0;
  var n:=0; var k:=1; var m:=6;
  while n<num
  invariant n<=num;
  invariant cube == n*n*n;
  invariant k == 3*n*n + 3*n +1;
  invariant m == 6*n + 6;
  {
    cube := cube + k;
    k := k + m;
    m := m + 6;
    n := n + 1;
  }
}
```

```
Dafny program verifier finished with 1 verified, 0 errors
```

```
method Main ()
{ var i := Cube(5); print i; }
```

```
Dafny program verifier finished with 2 verified, 0 errors
Program compiled successfully
Running...
125
```

We have also PROVEN this program will work as intended using simple algebra and expansion and Hoare logic, this is like when we check the pre/post of the Hoare triple.

No guessing!

Classes

Dafny has java and c# like classes

- Define complex models
- Dynamically allocate hidden and mutable data structures
    - No inheritance
- Objects are derived from the instantiation of a class using its constructor
    - Var obj = new obj()
- Verification becomes more difficult, we need to verify the class rather than a method

Classes have

- Methods
- Functions and predicates
- Constructor
- Class variables
- Class invariant

A static method can be called without creating an object of the class.

- Will not be able to access instanced methods

- No state

An instance method requires the object to be created before called

- Uses new keyword and constructor
- Deep copies
- Dynamic state

```
class Sumit                     // CLASS HAS NO SPEC!
{
   var a: int, b: int;          // class variables define state

   constructor (aa: int, bb: int) // an instance method that modifes state
   { a := aa; b := bb; }

   method Sum() returns (sum: int) // an instance method: doesn't modify state
   modifies this;                  // not necessary here, but Dafny won't complain
   { sum := a + b;}

   static method SumStatic(aa: int, bb: int) returns (sum: int)
   // modifies this             // there is NO 'this'! This would cause an error.
   { sum := aa + bb; }          // code cannot access class variables
}

method Main()
{
   var s := new Sumit(1, 2);         // s is an object (which has Sumit's state)
   var sum := s.Sum();               // sum the class variables
   print "state sum is ", sum, '\n'; // cannot use an assert as there is no spec

   sum := Sumit.SumStatic(1, 2);     // call Sumit's static method
   print "static sum is ", sum;
}
```

```
state sum is 3
static sum is 3
```

Just like java or c#, we don't need a constructor, we can create and modify with pre-defined values

- We can have problems with constructors
- Sometimes it might be better to use Init
    - Create object
    - Call Init

Constructors

```
class NameClass {
   constructor (…) // do not use a modifies clause in a constructor
   { … }

   method Init(…)
   modifies this;
   { … }
}
```

You can use `var n := new NameClass(…) // n is an object from the class`

or you can use `var n := new NameClass.Init(…) // n is an object from the class`

Ghosts

- Entities used in verification only
    - Functions and predicates ARE ghosts we've been using them all along
- Lemmas are ghost methods
- Ghost variables
    - Verifier will treat them as normal variables
    - Compiler will remove ghost variables and functions and lemmas and predicates once compilation occurs

Verifying classes

- We need a class invariant conventionally called Valid()
- Every method must hold the class invariant before and after the call
- A constructor will establish the class invariant AFTER constructing the object

Sometimes we use ghost variables to specify the behavior and create our invariant since this can be a complex task

- Ghost variables can shadow the program variables
- Class invariant links ghost and program variables

```
class Account // no ghost variables
{
  var balance: int;                  // class variable

  predicate Valid()                  // class invariant
  reads this;
  { balance >= 0 }

  constructor (val:int)
  requires val>0;
  ensures Valid();
  ensures balance==val;
  { balance := val; }

  method Withdraw(val: int)
  requires Valid(); ensures Valid();
  requires val>0 && balance>=val;
  modifies this;
  { balance := balance - val; }

  method Deposit(val: int)
  requires Valid(); ensures Valid();
  requires val>0;
  ensures balance>=0;
  modifies this;
  { balance := balance + val; }
}
```

What would happen if I commented this out?

```
Dafny program verifier finished with 3 verified, 0 errors
```

If we commented out the pre-condition of the constructor, we would get post condition might not hold, since if we pass in a negative value it would violate the post-condition

Using a ghost variable in creating a simple excel cell class

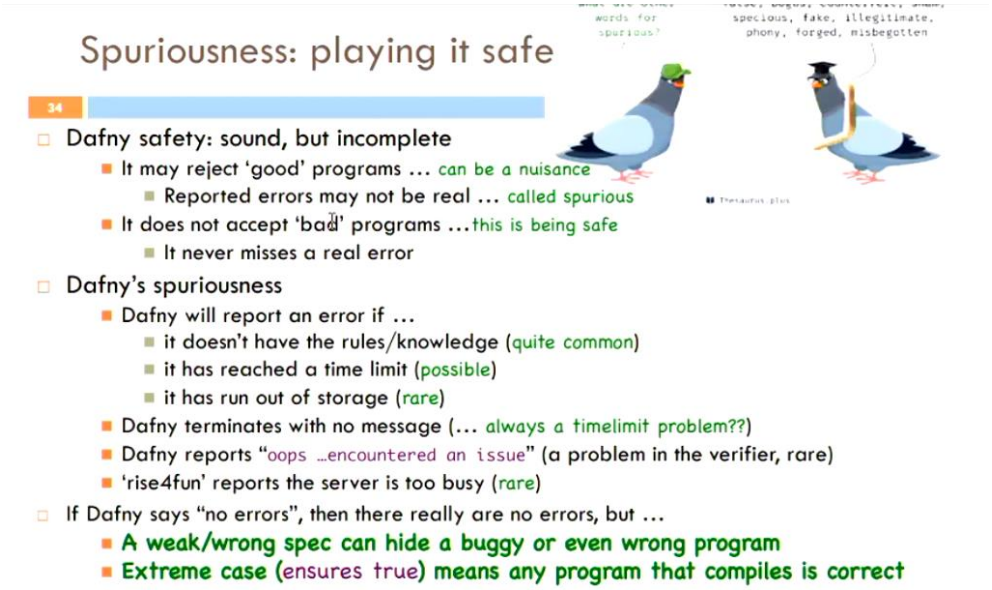
```
class Cell { // with ghost variables (not useful here)
  var data:int;
  ghost var gdata:int; // gdata 'shadows' data in this spec

  predicate Valid()
  reads this;
  { gdata == data } // link the program and ghost variables

  constructor (i:int)
  ensures Valid();
  ensures gdata==data==i;
  { data:= i; gdata := data; } // initialise the cell

  method Get() returns (r: int)  // modifies not required
  requires Valid();
  ensures Valid();
  ensures r==data;
  { r := data;  }

  method Set(newdata: int) returns()
  modifies this;
  requires Valid();
  ensures Valid();
  ensures gdata==data==newdata;
  { data := newdata; gdata := data; }
}
```

```
method VerifyCell(){
  var c := new Cell(-1);

  var i := c.Get();
  assert i==-1;    // tc1

  c.Set(42);
  i := c.Get();
  assert i==42;    // tc2
}
```

Here we use the ghost variable to ENSURE that what happens to our executed data can be applied to verification, it is our way of verifying updates in state.

```
method VerifyCell(){
  var c := new Cell(-1);

  var i := c.Get();
  assert i==-1;    // tc1

  c.Set(42);
  i := c.Get();
  assert i==42;    // tc2
}
```

Lecture 9



Modelling

We live in the ghost world; we will verify we won't be executing.

To do verification on classes we also need to use the old keyword as the old keyword will tell us the information about the state of the program BEFORE code is executed.

```
class ShowOld
{
  var a:int, b:int;

  method Swap()
  modifies this;
  ensures a == old(b)
  ensures b == old(a);
  { a, b := b, a; }

  method Another(c:int)
  modifies this;
  ensures a == old(a+b)+c;
  { a := a+b+c; }
}
```

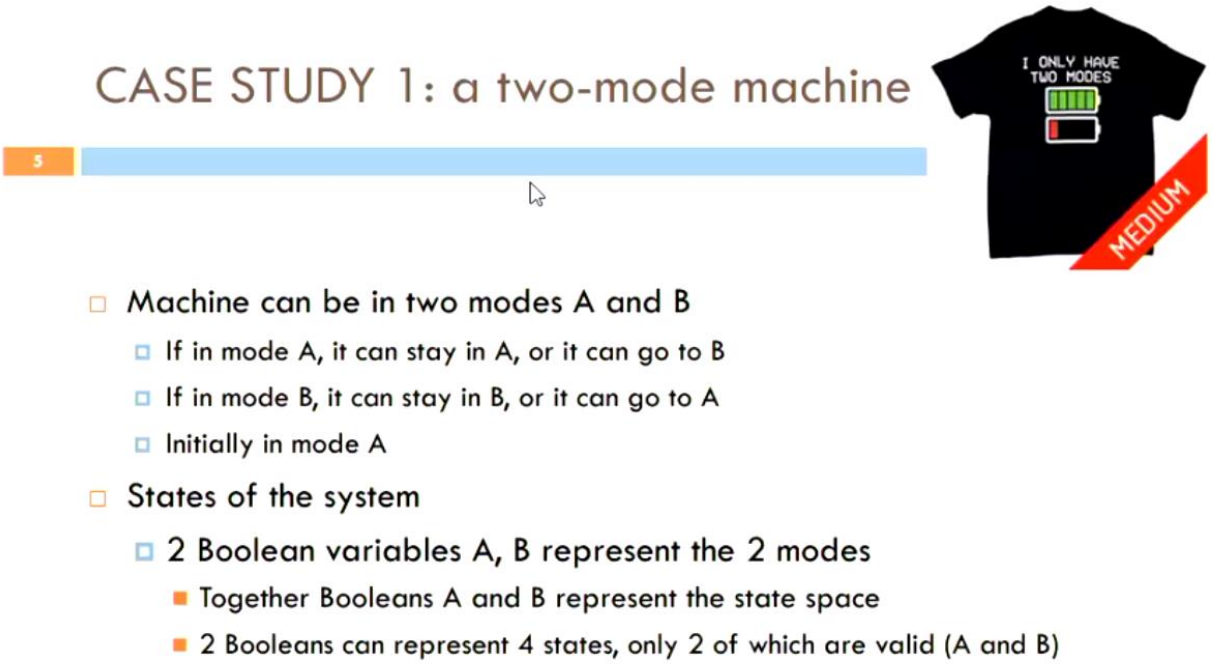
It would be difficult to do this if you could not call old().

In modelling we start with the view of the problem rather than the algorithmic view

- The model forms the basis of the design
- Development steps will be the decreasing level of abstraction
- The lowest level of abstraction will be implementation
- We look at models made up of data and events
- Data is represented by a small set of discrete states
    - Deterministic
        - Single and current state
- events or actions will change the state in a deterministic fashion

Example a t-shirt

Remember finite automata



Assumption is missing

- assuming there are only two modes
- we assume stay means go to where it is at
- we assumed stay means the same as go but in current state

in implementation

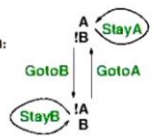
- we don't need two variables, since the actual state is either in A or B, so Boolean could be
    - we are in A

We want to say that you can't be in both states, but only in 1

State space:

Boolean A	Boolean B	comment
true	true	Not allowed
true	false	In mode A
false	true	In mode B
false	false	Not allowed

State transition diagram:



```
class Machine{
   ghost var A: bool; // 'ghost' because just verifying, not coding
   ghost var B: bool;

   predicate Valid()
   reads this
   { (A && !B) || (!A && B) } // defines the two feasable states

   constructor ()
   ensures Valid();
   ensures A && !B;   // start state is A
   { A := true; B := false;}

   method StayA()                          method GotoA()
   modifies this;                          modifies this;
   requires Valid(); ensures Valid();      requires Valid(); ensures Valid();
   requires A; ensures  A;                 requires B; ensures A
   { }                                     { A := true; B := false; }

   method StayB()                          method GotoB()
   modifies this;                          modifies this;
   requires Valid(); ensures Valid();      requires Valid(); ensures Valid();
   requires B; ensures  B;                 requires A; ensures B
   { }                                     { A := false; B := true; }
```

First 3 lines of every method should be this

```
method StayA()
modifies this;
requires Valid(); ensures Valid();
```

- every action is a method
- predicate calls read
- methods call modifies
- valid will
    - remove infeasible states
    - constructor establishes this predicate
- every method maintains valid

```
method TestOk()
{
   var m := new Machine(); // we know it starts in A
   m.StayA(); m.StayA();   // still in A
   m.GotoB(); m.StayB();   // now in B
   m.GotoA(); m.StayA();   // back to A
}
```

- Code it 'wrong' and you'll get a verification error

```
method TestNotOk()
{
   var m := new Machine(); // this will start in A
   m.GotoA();              // will cause an error
}
```

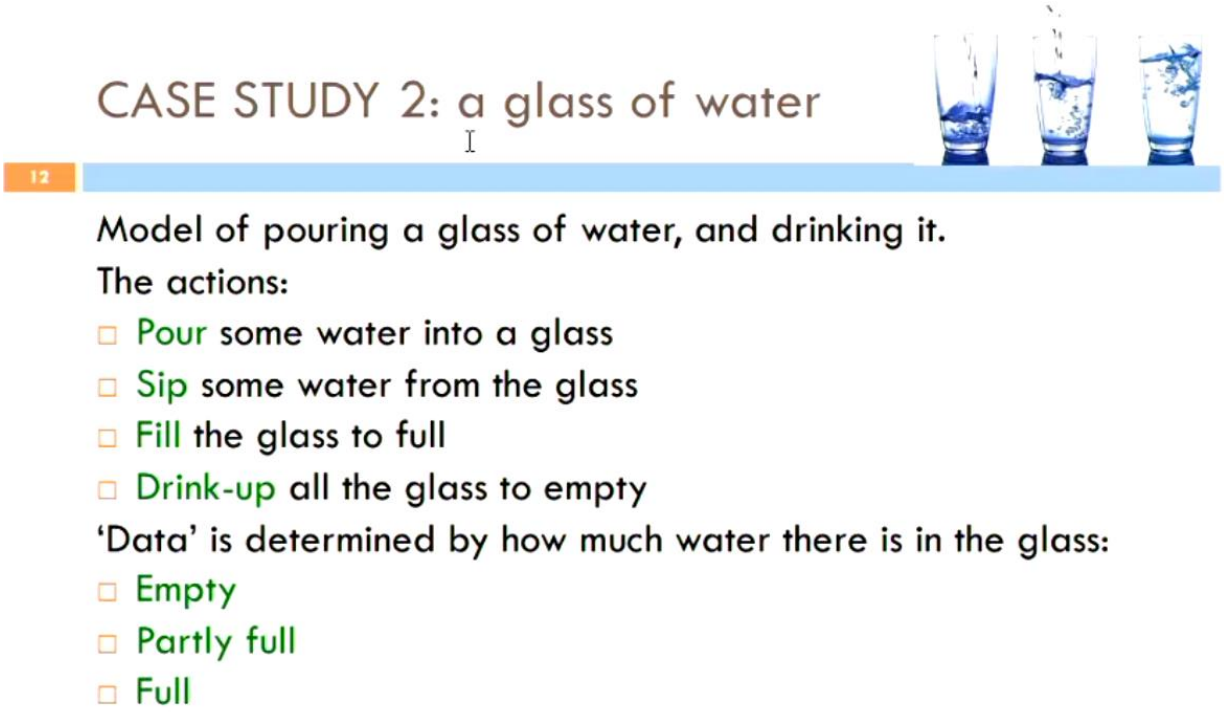
The testnotok will fail because you're already at A you need to call stay A, it requires B

This is a language error

Class reminders

- a state transition diagram is essential for understanding the model and its behavior
- there is natural 'symmetry' in the diagram
- the state variables are ghosts
- class invariant defined class properties
    - stronger the invariant ➔ smaller state space
- a constructor will establish your class invariant
- every action maintains the class invariant
- multiple test methods
- show bad behavior being picked up by the verifier

Making dafny drink water



Note we differentiate with pour and fill, and sip and drink-up

- fill can be seen as pour till glass is full
- drink up can be seen as sip till glass is empty

here our assumptions are

- there is a difference between pour and fill
- difference between sip and drink-up

State of the glass:

- Like two-mode machine, 2 Boolean variables
  - IsEmpty, IsFull
- 2*2 possible states, 3 feasible states representing 'data'
  - empty: IsEmpty ∧ !IsFull
  - full: !IsEmpty ∧ IsFull
  - partly full: !IsEmpty ∧ !IsFull

State space

IsEmpty	IsFull	comment
true	true	impossible
true	false	glass is empty
false	true	glass is full
false	false	glass is partly empty/full

State transition diagram


```
class Glass{
   ghost var IsEmpty: bool;
   ghost var IsFull: bool;

   predicate Valid()
   reads this
   { !(IsEmpty && IsFull) } // can't be empty and full at the same time


constructor()
ensures Valid();
ensures IsEmpty && !IsFull;   // start with an empty glass
{ IsEmpty := true; IsFull := false; }
```

```
method Pour()
 modifies this;
 requires Valid(); ensures Valid();
 requires !IsFull;//cannot pour into full
 ensures !IsEmpty && !IsFull; //cannot fill
 { IsEmpty := false; IsFull := false; }

 method Fill()
 modifies this;
 requires Valid(); ensures Valid();
 requires !IsFull; //cannot fill full
 ensures  !IsEmpty && IsFull;
 { IsEmpty := false; IsFull := true; }
```

```
method Sip()
modifies this;
requires Valid(); ensures Valid();
requires !IsEmpty // cannot sip empty
ensures  !IsEmpty && !IsFull; // cannot empty
{ IsEmpty := false; IsFull := false; }

method Drinkup()
modifies this;
requires Valid(); ensures Valid();
requires !IsEmpty; // cannot drink empty
ensures  IsEmpty && !IsFull;
{ IsEmpty := true; IsFull := false; }
```

- what a different way to think of drinking water
- makes sense, abstracts away the notion of volume
- describes the behavior at the highest level

Testing

```
method VariousTests()
{
 var g:=new Glass();                          // each scenario terminates with drinkup
 g.Pour(); g.Pour(); g.Fill(); g.Drinkup();       // 1. pour slowly, drinkup when full
 g.Fill(); g.Drinkup();                           // 2. fill glass, drinkup
 g.Pour(); g.Drinkup();                           // 3. partial fill glass,drinkup
 g.Pour(); g.Sip(); g.Sip(); g.Sip(); g.Drinkup(); // 4. pour some, sip, finally drinkup
 g.Pour(); g.Sip(); g.Pour(); g.Sip(); g.Drinkup();// 5. pour, sip repeatedly, drinkup
}
```

Tests that should and do fail:

```
 var g:new Glass(); g.Drinkup;           // 6. error, can't drinkup from empty glass
 var g:new Glass(); g.Sip();             // 7. error, can't sip from an empty glass
 var g:new Glass(); g.Fill(); g.Pour();  // 8. error, can't pour when full
 var g:new Glass(); g.Fill(); g.Fill();  // 9. error, can't fill when full
```

We can also show our state as an integer representing state

- 1 == full
- 2 == empty
- 3 == partially full

We can now rewrite the spec to say that

```
predicate Valid()
reads this
{ amount==0 || amount==1 || amount==2 } // the 3 feasible states
```

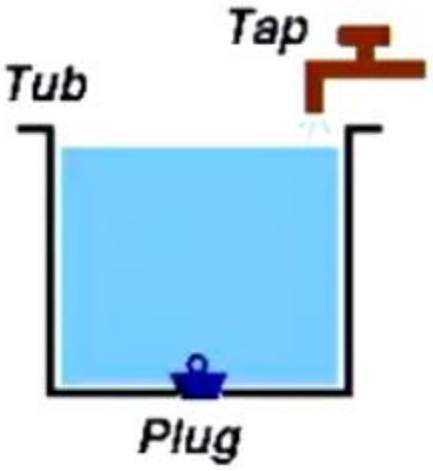
And now everything flows logically form there

Case bathtub

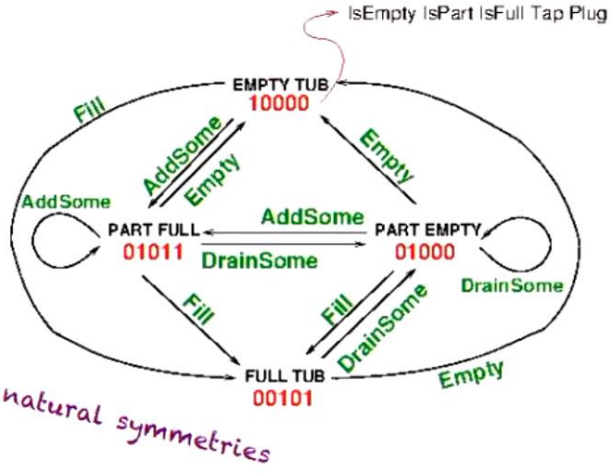
The state space of Tap and Plug

Tap	Plug	comment
true (on)	true (in)	add some water to tub
true (on)	false (out)	not allowed
false (off)	true (in)	tub is full
false (off)	false (out)	draining, or tub is empty

- We can add a special cleaning mode where we have tap on plug out, to allow for this one case



State transition diagram



We almost have the dafny spec right there, almost automatic to generate the code to satisfy this specification

```
class Tub
{
  ghost var IsEmpty: bool;
  ghost var IsPart: bool;
  ghost var IsFull: bool;
  ghost var Tap: bool;   // on == true, off == false
  ghost var Plug: bool;  // in == true, out == false

  predicate Valid()
  reads this
  {((IsEmpty && !IsPart && !IsFull) ||
    (!IsEmpty && IsPart && !IsFull) ||
    (!IsEmpty && !IsPart && IsFull)) // mutually exclusive
    && !(Tap && !Plug)       // be safe: don't allow tap on and plug out
    && !(Plug && IsEmpty)    // be safe: don't allow plug in when tub empty
    && !(Tap && IsFull)      // be safe: don't allow tap on and full tub
  }
```

Here is where you decide what are you missing? What states do you want to allow/disallow, this is almost like our specification crux

```
constructor()
ensures Valid();
ensures IsEmpty && !Tap && !Plug // start is empty tub, tap off, plug out
{
   IsEmpty := true;
   IsPart, IsFull, Tap, Plug := false, false, false, false;
}

method AddSome()  // add some water (cannot fill the tub)
modifies this;
requires Valid(); ensures Valid();
requires IsEmpty || IsPart;
ensures IsPart && Tap && Plug; // tap on, plug in
{
  IsPart, Tap, Plug := true, true, true;
  IsEmpty, IsFull := false, false;
}
```

```
method Fill() // fill the tub to the top
modifies this;
requires Valid(); ensures Valid();
requires IsEmpty || IsPart;
ensures  IsFull && !Tap && Plug; // turn the tap off!
{ IsFull, Plug := true, true; IsPart, IsEmpty, Tap := false, false, false;}

method DrainSome() // drain off some water (cannot empty the tub)
modifies this;
requires Valid(); ensures Valid();
requires IsPart || IsFull;
ensures IsPart && !Tap && !Plug; // tap off and pull the plug
{ IsPart := true; IsEmpty, IsFull, Tap, Plug := false, false, false, false;}

method Empty() // empty the tub to the bottom
modifies this;
requires Valid(); ensures Valid();
requires IsPart || IsFull;
ensures  IsEmpty && !Tap && !Plug; // tap off and pull the plug
{ IsEmpty := true; IsPart, IsFull, Tap, Plug := false, false, false, false;}
```

We can even add temperature is Hot is Cold is Warm, and this would greatly complicate the model,

```
method TestTub()
{
var t := new Tub();
t.AddSome(); t.AddSome(); t.AddSome(); t.DrainSome(); t.Fill(); // 1. add/fill the tub
t.DrainSome(); t.AddSome(); t.DrainSome(); t.Empty();// 2. drain/add and finally empty
t.Fill(); t.Empty();                                  // 3. fill and empty
t.AddSome(); t.Empty();                               // 4. add and empty
// t.Empty();                              // 5. error: you cannot empty an empty tub
// t.DrainSome();                          // 6. error: you cannot drain an empty tub
// t.Fill(); t.AddSome();                  // 7. error: you cannot add to a full tub
// t.Fill(); t.Fill();                     // 8. error: you cannot fill a full tub
}
```

```
Dafny program verifier finished with 7 verified, 0 errors
```

Lecture 10

Multisets

- Like a set but allows duplicate elements
- Groups all elements with same key
- Allows you to count occurrences of keys
- Union is different to set and subtract
    - Adds elements with duplicates unlike normal sets
- Can be constructed with a sequence
- All other set operations occur as expected

```
method Main() // playing with multisets
{
    var a:= new int[][2,4,6,8];                    // this is an array of course
    var ms:multiset<int> := multiset(a[..]);       // create a multiset from the array
    assert 2 in ms && 5 !in ms;                    // look inside
    assert |ms|==4;                                // check the length

    var ms1 := ms + multiset([2,4]) + multiset{2}; // union of 3 multisets
    print ms1, '\n';
    var ms2 := multiset{4} + multiset([2,2]) + ms; // same union, different order
    print ms2, '\n';
    print "Notice the multiset orders are not what you expect\n";
    print "Are they equal? ", ms1==ms2, '\n';      // check the output

    var i:=0;
    var v:=10;
    var ms3 := multiset(a[..][i:=v]);              // more creation, with an update
    var ms4 := multiset(a[..][0:=10][1:=20]);      // creation with a double update
    print ms3, '\n';
    print ms4, '\n';
}
```

==multiset{2,4,6,8}

update element at index i with value v

```
Dafny program verifier finished with 1 verified, 0 errors

multiset{2, 2, 2, 4, 4, 6, 8}
multiset{4, 4, 2, 2, 2, 6, 8}
Notice the multiset orders are not what you expect
Are they equal? True
multiset{10, 4, 6, 8}
multiset{10, 20, 6, 8}
```

```
method Main()
{
    var a := multiset{'C','A','B','B','A'};
    var b := multiset{'C','E','D','G','E'};
    var c := multiset{'A', 'B'};

    print "a = ", a, '\n';
    print "b = ", b, '\n';
    print "c = ", c, '\n', '\n';
    print "addition: a+b = ", a+b, '\n';
    print "difference: a-b = ", a-b, '\n';
    print "intersection: a*b = ", a*b, '\n';
    print "disjointedness: a!!b = ", a!!b, '\n';
    print "disjointedness: b!!c = ", b!!c, '\n';
    print "comparison: c<a = ", c<a, '\n';
    print "comparison: c<b = ", c<b, '\n';
}
```

In the Dafny Reference manual, multiset '+' is called *union*.

Mathematically, in the *union* of two multisets, each member has the maximal multiplicity it has in either of its operands.
Notice that '+' has two C's here, so it really is addition

```
Running...
a = multiset{C, A, A, B, B}
b = multiset{C, E, E, D, G}
c = multiset{A, B}

addition: a+b = multiset{C, C, A, A, B, B, E, E, D, G}
difference: a-b = multiset{A, A, B, B}
intersection: a*b = multiset{C}
disjointedness: a!!b = false
disjointedness: b!!c = true
comparison: c<a = true
comparison: c<b = false
```

Inductive Haskell Types

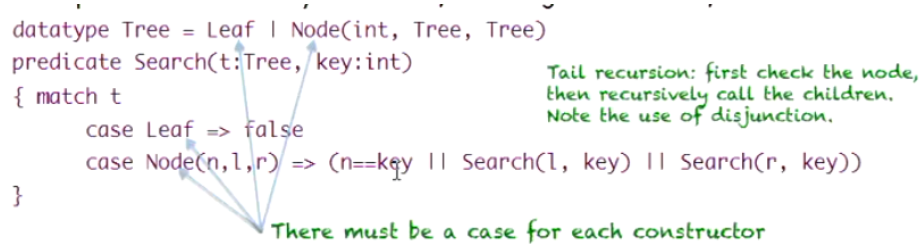
Inductive data types provide recursive structures

- Datatype Trees = Empty | Node(int, left, right)
- Datatype Lists = Empty | data + List[]

We can also name the parameters of our inductive data types

- Datatype Tree = Empty | Node(data: int, left: Tree, right: Tree)

Example



Note I prefer the syntax

Match(t){

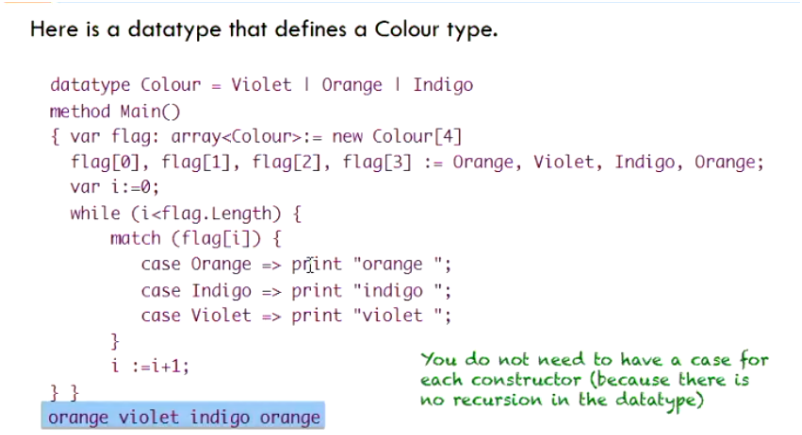
Case Leaf => false

Case Node(n, l, r) => (n == key || Search(l, key) || Search(r, key))

}

This is just a simple switch statement in most program languages that allow for pure recursion, base case and recursive case

Another example of using match on a base type



In this case, if we don't need to consider orange/indigo since they are atoms, only on recursive constructors we need a case

Here we will define what a tree is as either

- 0 children leaf
- 1 child node
- 2 child nodes

And we will write a method that prints it using the inductive definition

```
datatype Tree<T> = Leaf(T) | Node1(T, Tree<T>) | Node2(T, Tree<T>, Tree<T>)

method Print(t:Tree<int>) // BOTTOM-UP
{ match t // 3 constructors means need 3 cases
    case Leaf(n) => print n, ' ';
    case Node1(n,l) => { Print(l); print n, ' '; }
    case Node2(n,l,r) => { Print(l); Print(r); print n, ' '; }
}
method Main(){
  var leaf1:Tree<int> := Leaf(1);
  var leaf2:Tree<int> := Leaf(2);
  var leaf3:Tree<int> := Leaf(3);
  var leaf4:Tree<int> := Leaf(4);
  var t5:Tree<int> := Node2(5, leaf1, leaf2);  // Binary tree
  var t6:Tree<int> := Node2(6, leaf3, leaf4);  //       7
  var t7:Tree<int> := Node2(7, t5, t6);        //   5       6
  Print(t7); print '\n';                       // 1   2   3   4
}
verifier finished with 2 verified, 0 errors          Running...  1 2 5 3 4 6 7
```

A method to find height of tree

Inductive datatypes

1) The height of the tree

- here use a function method so the result can be printed

```
function method Height(t:Tree<int>):nat
{ match t
   case Leaf(n) => 1 // leaf node is at height 1
   case Node1(n,l) => 1 + Height(l)
   case Node2(n,l,r) => 1 + Max(Height(l), Height(r))
}
function method Max(i:nat, j:nat):nat
{  if i>j then i else j }
```

- Testcase (add to Main()):

```
var i := Height(t7);
print "Height = ", i, '\n';
```

```
Height = 3
```

We can even do crazy stuff like store the tree into a multiset

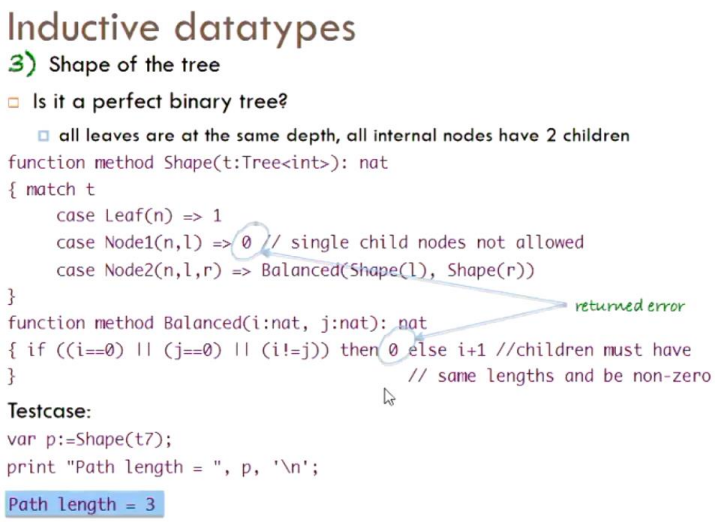
```
function method Bag(t:Tree<int>):multiset<int> // TOP-DOWN
{ match t
    case Leaf(n) => multiset{n}
    case Node1(n,l) => multiset{n} + Bag(l)
    case Node2(n,l,r) =>  multiset{n} + Bag(l) + Bag(r)
}
```

Testcase (add to Main()):

```
var b:multiset<int> := Bag(t7);
print "Bag = ", b, '\n';
```

```
Bag = multiset{7, 5, 1, 2, 6, 3, 4}
```

We can even check if a tree is perfectly shaped (all nodes fully internal)



We can write a method to check every node inside a tree is 0

```
predicate AllZero(t:Tree<int>)
{ match t
      case Leaf(n) => n==0
      case Node1(n,l) => (n==0 && AllZero(l))
      case Node2(n,l,r) => (n==0 && AllZero(l) && AllZero(r))
}
```

We can also write a predicate to check if a tree is a perfect binary tree, perfectly balanced where every node is fully internal

```
predicate Perfect(t:Tree<int>, dep:nat)
{ if dep == 0 then match t
                  case Leaf(n) => true
                  case Node1(n,l) => false
                  case Node2(n,l,r) => false
              else match t
                  case Leaf(n) => false
                  case Node1(n,l) => false
                  case Node2(n,l,r) =>(Perfect(l,dep-1) && Perfect(r,dep-1))
}
```

Depth == 0 means we are at a leaf, if not we return false, otherwise we want to only do the check IF the node is fully internal

We can put it all together to get this

6) A predicate checking whether the tree is perfect and all zero

```
predicate PerfectZero(t:Tree<int>, n:nat)
{ Perfect(t,n) && AllZero(t) }
now the initialisation method (notice the post-condition)
method Init(t:Tree<int>) returns (newt:Tree<int>)
ensures forall m:nat:: Perfect(t, m) ==> PerfectZero(newt, m);
{ match t
      case Leaf(n) => {newt := Leaf(0);}
      case Node1(n,l) => { var x := Init(l);
                           newt := Node1(0, x); }
      case Node2(n,l,r) => { var x := Init(l);
                             var y := Init(r);
                             newt := Node2(0,x,y); } }
```

Sub-tree passed up the tree

Testcase:

```
34 t7 := Init(t7); Print(t7);          0 0 0 0 0 0 0
```

Note that our post condition says, that if we have a perfect tree, we will return a perfectly zerod tree

We can even check this all works by checking a counter example

- Add a new node 42 as child to the last leaf 4, and re-link it upwards

```
var leafx:Tree<int> := Leaf(42);                    //        7
leaf4 := Node1(4, leafx);  // was 'leaf4 := Leaf(4)'      5      6
t6 := Node2(6, leaf3, leaf4);                       //   1   2  3   4
t7 := Node2(7, t5, t6);                             //              42
Print(t7); print '\n';
```

```
1 2 5 3 42 4 6 7
```

This is the 'bottom-up' position for the new leaf: between 3 and 4

- We can see the tree is no longer perfect: but what does 'Shape' say?

```
var p:=Shape(t7);
print "Path length = ", p, '\n';
```

```
Path length = 0
```

This is the error condition: the tree is 'imperfect'

Dafny Data Structures

BIC BOI CONTENT

Normally in computer science we use data structures to hold and manipulate data in a behavioral way, example Queue will have push/poll/peek and we always use FiFO

Example stack will have push/pop/peek and we always use FILO

But how do we verify the data structure is correct?

- We ensure that all its methods are correct
- And that its class invariant is maintained

Two kinds of verification of high-level data structures:

- Program verification on the data structure code:
  - Treats the data structure operations as program code
  - Verify using post-conditions that specify the effect of the operation
  - Valid() is a pre- and post-condition of every operation
  - We will see two examples of this: Hardstack and Cyclestore
- Ghost verification on a ghost data structure, typically called *shadow*
  - Verify using post-conditions that specify the effect of the operation on the shadow
  - Valid() is used to specify the relationship between shadow and the actual data structure
  - As with program verification, Valid() is maintained by every method
  - We will see two examples of this: Quack and Queue

The key difference between program and ghost verification is that program verification will verify the methods behave correctly, where ghost verification shadows the data structure and will check the effect on the shadow

With all data structures we use low level data structures to manage it

- Arrays (most common)

The operations are like an api that are high order, and the data is the low order behind the scenes

- We need to verify all the operations

HardStack

Fields of our stack

```
var buf: array<int>;  // array of contents
var top: int;         // index of top of stack
var cap: int;         // size of the stack
```

Class invariant

```
predicate Valid()
reads this;
{
   0<cap<=buf.Length &&  // cap fits into array
  -1<=top<cap            // top is valid
}
```

The class invariant says that the cap of the array must be between array bounds (will always be buf.length to be honest since we set it to that when we initalise)

It also says that the top will be between -1 and cap (between empty and full)

- Push and pop happen at index top
- An empty stack is defined to be top==-1

It is a hard stack because, if we push an element onto a full stack::

- we make room by deleting the oldest element
- push down every element
- push the new element (so stack is again full)

```
class HardStack {
   var buf: array<int>;  // contents
   var top: int;         // index of top of stack, -1 if empty
   var cap: int;         // capacity of the stack

   predicate Valid()
   reads this;
   { 0<cap<=buf.Length &&
    -1<=top<cap
   }

   method Init(size: int)
   modifies this;
   requires size>0;
   ensures Valid(); //establish invariant
   ensures top==-1 && cap==size;
   ensures fresh(buf); // tell the world buf was created by Init(), see later
   { buf := new int[size];
     top := -1;
     cap := size;
   }

   predicate isEmpty() // used in Main()
   reads this;
   { top == -1 }

   predicate isFull() // used in Main()
   reads this;
   { top == cap-1 }
```

Notice we use fresh on Init, which states that when the array wrapper is created it is clean and a NEW array

Very important especially for the performance and sake of our sanity

- Only say we are modifying what we are actually modifying, if all we are modifying is the top, only state modifies this `top
    - If we say modifies this, then it is our obligation to prove everything else stays the same, we put more work on ourselves
- If a method doesn't change anything do not say modifies this

```
method Peek() returns (elem: int)  // peek at the top, without removing it
requires Valid(); ensures Valid(); // maintain invariant but redundant
requires 0<=top<cap;               // stack can't be empty
ensures elem == buf[top];          // postcond matches the code
{ elem := buf[top]; }

method Pop() returns (elem: int)   // pops top element off non-empty stack
modifies this`top;                 // only top is changed, which is not an object, use `
requires Valid(); ensures Valid();
requires 0<=top<cap;               // cannot be empty
ensures elem == buf[old(top)];     // postconds specify what the code does
ensures top == old(top)-1;         // notice use of old()
{ elem := buf[top]; top := top-1; }

Keep the 'modifies' tight
```

Also note the use of old, it allows us to verify what happens when we modify the data structure, it allows us to easily almost simply go between our previous state and new state (just like in SQL)

- Notice that we don't modify the array since pop doesn't actually remove the element from the array it just returns it to the user and moves the stack down (top := top – 1)

It is so important to state EXACTLY what is happening in your post conditions, we need to say what happened to our state AFTER our method (use of old)

Adding in the functionality of our force push in our hard stack

The HardStack will allow us to call force push which is only to be called when the stack is full

- It will push EVERY element down by 1, and assign the new element to the top
    - Effectively erases the oldest element in stack
    - Stack size won't change

```
// Push an element onto the non-full stack
method Push(elem: int)
modifies this`top, this.buf;tight modifies: top is not an obj, so use `  buf is an obj, so use .
requires Valid(); ensures Valid();
requires -1<=top<cap-1; the stack is not full
ensures forall i:: 0<=i<=old(top) ==> buf[i]==old(buf[i]);
ensures 0<=top<cap;
ensures buf[top]==elem;
ensures top == old(top)+1;
{ top := top+1; buf[top]:=elem; }

// Push by force onto full stack, bottom of stack element is overwritten
method PushX(elem: int)
modifies this.buf; tight modifies
requires Valid(); ensures Valid()
requires top==cap-1; the stack is full
ensures forall j:: 0<j<=top ==> buf[j-1]==old(buf[j]);
ensures buf[top]==elem;
{
   forall (i| 0<i<=top) { buf[i-1]:=buf[i]; }
   buf[top] := elem;
}
```

Notice, a 'ghost' forall is used to verify an 'executable' forall

executable forall statement

What's that?

Notice we aren't changing modifying top since the stack stays the same size. But also, we are using a forall quantifier almost as like a loop (IT IS NOT A LOOP) it almost is like a foreach loop. The second forall is an executable forall it is not like the predicate forall

Tests on our HardStack

```
method StackVerify() {
  var s := new HardStack.Init(3);
  assert s.isEmpty() && !s.isFull();

  s.Push(10);
  assert !s.isEmpty() && !s.isFull();

  var f := s.Peek();
  assert f == 10;
  assert !s.isEmpty();
  f := s.Pop();
  assert f == 10;
  assert s.isEmpty();
```

```
s.Push(10);s.Push(20);s.Push(30);
assert s.isFull();
f := s.Pop();
assert f == 30;
f := s.Pop();
assert f == 20;
f := s.Pop();
assert f == 10;
assert s.isEmpty();
```

```
s.Push(10);s.Push(20);s.Push(30);
s.PushX(40);
f := s.Pop();
assert f == 40;
f := s.Pop();
assert f == 30;
f := s.Pop();
assert f == 20;
assert s.isEmpty();
```

```
}
```

```
Dafny program verifier finished with 13 verified, 0 errors
```

Also note that we can have the default push be push as normal, and if stack is full use push X. This will complicate the program and almost forces us to use a smart program to check the stack before it chooses which push to use. We have shown here how to do it separately.

Executable foralls

- A parallel assignment statement, all occurs in 1 go
- The body can only be a single assignment statement or method call
- Can't print inside
- Not an actual loop but is almost behaved like a loop in some cases
- Finite
- No invariant, but post condition

The general syntax is:

```
forall x: T | R(x,y) ensures P(x,y) { body }
```

where *x* is bound here, and *y* is not. Can also have more bound variables

```
forall x:T, y:U | R(x,y,z) ensures P(x,y,z) { body }
```

where *x* and *y* are bound here, and *z* is not.

Actually a generalisation of universal quantification

- The body is used to prove the post-condition *P(x)*
- Is similar to a parallel assignment statement (used to be called 'parallel')
- Not a loop (but sometimes used to mimic a loop).
- Must be finite.
- The body consists of ghost statements (could be lemmas)
  - Can be just 1 assignment statement (to an object) or a method call ('print' not allowed)
- Note that no invariant is required (because it is not a loop)

Example of cycling an array

- `forall i | 0<=i<a.Length { a[i] := a[(i+1)%a.Length]; }`
  - does a *left rotation* of the array elements
  - remember, "reads before write" (so *a[0]* is not 'lost')
  - *a[0]:=a[1]; a[1]:=a[2]; … a[a.Length-1]:=a[0]; similtaneously*

This also allows us to do very easy post condition assurance for our methods.

Fresh

Fresh is used to check initialization occurred correctly.

- In cases sometimes initalisation can go wrong and have drastic consequences
- Let's say our initalisation gives us sensitive information, this is catastrophic.
- We ensure fresh on initlisation, so we know that all our initilisation occurs correctly.
- Dafny doesn't know if we initialized our array it only knows its methods post condition
- So ensures fresh is a shorthand way of saying
- Ensures a = new array<T>()

Dafny has the notion of a *frame*

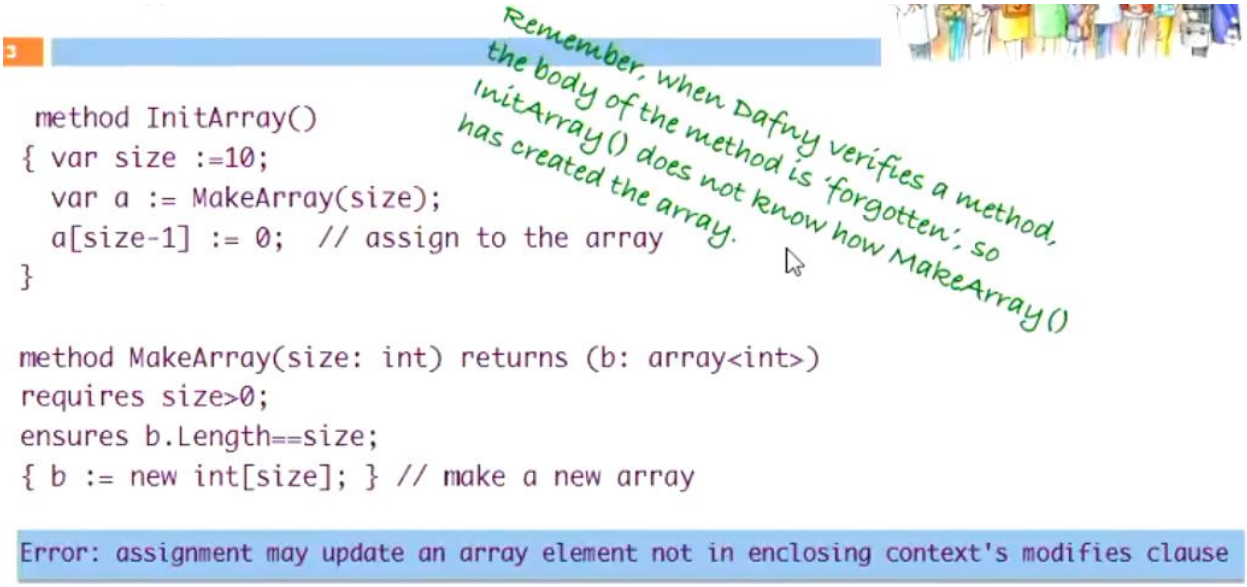
- A method's *frame* is the external data that the method is allowed to modify
  - hence `modifies`
- A function's *frame* is the external data that the function is allowed to read
  - hence `reads`

This notion is used for efficiency purposes only:

- Proving a method is correct for all the data in the program could take a very long time (hours, days, …)
- A modifies clause restricts the amount of data that needs re-verification in every execution path through the program.

Frames are important in classes:

- class fields must be included in a modifies clause if they are modified
- If a method tries to modify a field not in its modifies list, the result is:
  `Error: call may violate context's modifies clause`
- Local data in a method is always allowed to be modified



Lecture 11

- To handle OO which usually modifies during run-time, Dafny will try to predict what can occur from its methods
    - Specifying objects is still a very hard and current problem
    - Poorly understood from research point of view
    - Still current work in process to verify object oriented

Cycle Store

It just goes in a circle assigning, once it reaches the end it wraps around the array and starts from the beginning again

The invariant will ensure the cap is the size of the array, and the index is within the bounds of the array

```
class CycleStore
{
   var buf: array<int>; // a storage array
   var cap: int;        // capacity of array
   var ind: int;        // index to store an element

   predicate Valid()    // class invariant
   reads this;
   { cap>0 && cap==buf.Length && -1<=ind<=cap-1 }

   method Init(size: int)
   modifies this;
   requires size > 0;
   ensures Valid();
   ensures cap==size && buf.Length==size && ind==-1
   ensures fresh(buf);          // almost always needed in an Init() method
   {
     buf := new int[size];
     cap := size;
     ind := -1;                 // start index
   }
```

Now we need to write the method to push values in

```
  method Push(elem: int)       // store 'elem' at index ind+1
  modifies this.buf, this`ind;
  requires Valid(); ensures Valid();                                  // 1
  ensures if old(ind)==cap-1 then ind==0 else ind==old(ind)+1;        // 2
  ensures buf[ind]==elem;                                             // 3
  ensures forall i:: 0<=i<cap && i!=ind ==> buf[i]==old(buf)[i];      // 4
  {
     ind := if ind==cap-1 then 0 else ind+1; // this is where cycling happens
     buf[ind] := elem;
  }
} // end of class CycleStore
```

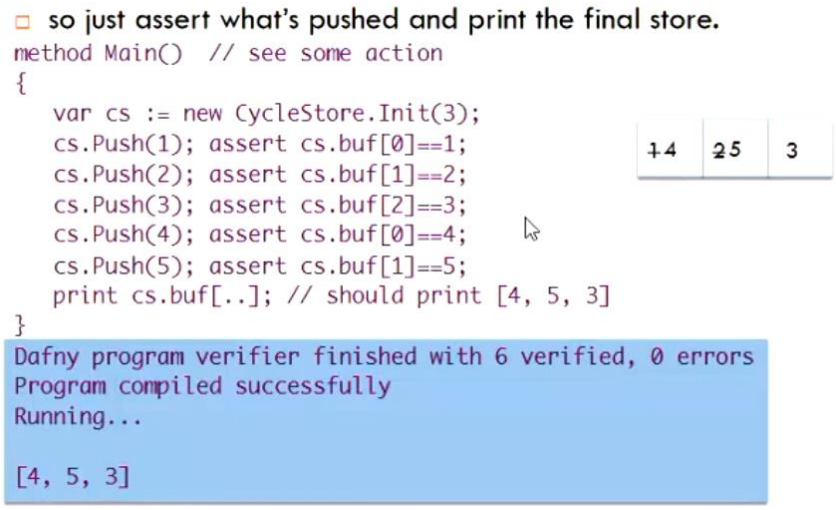
In this case, our post condition is just mirroring our code, very simple

Notice our post condition on 4 will also say what hasn't changed, since we say that the rest of the array in buf doesn't change, we need to specify that in our post condition. We also should ensure the cap doesn't change, we should say what isn't changing

An example of where it is crucial to say what hasn't changed is the swap within a sort

- If the programmers post condition just says the two elements are swapped, someone can swap the two elements and wipe the rest of the array
- Your program would verify since it doesn't check the rest of the array
- SPECIFY WHAT DOESN'T CHANGE

We could also add the same peek and pop, but that's unnecessary



(black box testing)

Used program verification no shadowing occurs. We could also emit cap since its useless since it's the length of the array, however it is more useful for ghost verification.

Program verification focuses on high level operations

- Post condition describes what the operation does

Maintains correctness using valid

- Valid must be true in the pre and post condition of every operation

Ghost verification

We try to shadow the real data and verify the effects on the ghost data

- The invariant valid will be used to link the real data with the ghost data

Ghost data which mirrors the original data can be tricky since we can't prevent errors such as ariane 5.

- If the real data has problems the ghost data will too

Quack

Can be either a queue or stack

Stack → pop

Queue → Qop

Operations

- Empty
- Pop
- Qop
- Push data

```
Empty(), Pop(), Qop(), Push(d)
```

An array is the backbone of this data structure where we actually store the data

- We limit the lower and upper bound
    - We use both m and n as our bounds to check the top and bottom of the array
        - Stack operations → n modify
        - Queue operations → m modify
        - When we push it's the same operation, but we modify n and m differently when we push to take out the element the way we want to
            - We keep m pointing to top, n pointing to bottom

```
var buf: array<Data>
var m:int, n:int; // part of array that is the quack
```

We will also use a sequence data structure to verify the quack operations

- The sequence called shadow is a ghost variable
- Shadow has the same data the array has
- Remember sequence is immutable
- Used only for verification

```
ghost var shadow: seq<int>;
```

QUICK IMPORTANT ASIDE

Dafny offers many class/method ***attributes***

- Many are directives to the underlying theorem prover
- Some affect the performance of Dafny directly

See the Ref. Manual:

§24.1.2 `:autocontracts` (concern frames and class invariant)

§24.1.6 `:fuel` (concerns depth of recursion)

§24.1.16 `:timelimit` and `:timeLimitMultiplier`

- Visual studio: time limit is 10 seconds (*not sure if same in 2.3.0*)
- *rise4fun*: more than 10 seconds
    - BUT hard time limit is 30 seconds (may result in no output)

These are settings we can change to modify how dafny works

We can set up an auto contract to do stuff we always do, e.g. requires + ensures valid

I'd rather make a snippet

Auto contracts not recommended

Usage:

```
class {:autocontracts} ClassName
```

- User must:
  - Declare a function/predicate `Valid()`
- Dafny does:
  1. Handles static and dynamic frames
     - `reads` and `modifies` in every function/method
     - declares a (dynamic frame) set of objects called *Repr*
       - access management for objects
  2. Adds the statement `ensures Valid()` to every constructor/method
  3. Adds the statement `requires Valid()` to every method

```
class {:autocontracts} Quack<Data(0)> //(0) means 'Data' can be initialised by Dafny
{
    var buf: array<Data>;
    var m: int, n: int; // indexes in buf[]

    ghost var shadow: seq<Data>;

    predicate Valid()
    { buf.Length!=0 && 0<=m<=n<=buf.Length && shadow==buf[m..n] }

    constructor(size: int)
    requires size>0;
    ensures shadow == [];
    ensures fresh(buf);
    {   buf := new Data[size];
        m, n:= 0, 0;
        shadow:= [];
    }
```

Notice, a generic data type

The actual queue is given by the two indexes

'shadow' shadows the actual queue

In this case study, a constructor will be used instead of a method (it was called Init() in the first two case studies).

The method `Empty()` tests whether the array is empty

- Dafny must verify the shadow (the sequence) is empty too

```
method Empty() returns (x:bool)
ensures x <==> shadow==[]      // x==true is equiv to an empty shadow
{
    x := m==n;                 // x is true if m and n are the same
}
```

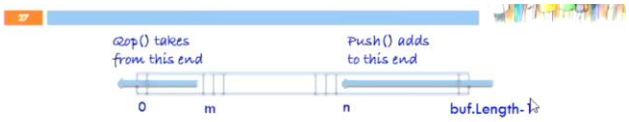
This is verification

This is code

WOULD BE BETTER TO SAY ISEMPTY IS THE NAME OF THE METHOD BY THE WAY

Also forgot requires valid ensures valid

```
method Pop() returns(x: Data) // + Push() together make a stack
requires shadow != []
ensures       x == old(shadow)[|old(shadow)|-1]   // this is the tail element
ensures  shadow == old(shadow)[..|old(shadow)|-1] // this is the new shadow
{
    x, n:= buf[n-1], n-1;                         // get tail, remove from buf
    shadow:= shadow[..|shadow|-1];                // chop the tail off shadow
}
```




```
method Qop() returns(x: Data)         // + Push() works as a queue
requires shadow != [];
ensures       x == old(shadow)[0]     // this is the head
ensures  shadow == old(shadow)[1..]   // this is the new shadow
{
    x, m := buf[m], m+1;              // get head and remove from buf
    shadow:= shadow[1..];             // chop the head off shadow
}
```

Wrong push, its too simple and since n keeps increasing we can go over the array bounds, its badly written this push

```
method Push(x: Data)
ensures shadow == old(shadow) + [x]  // easy to prove, you would think
{
   a[n], n := x, n+1;                // add new data to the array
   shadow := shadow + [x];           // add new data to shadow
}
```

- generates `Error: index out of range` on the first line of the method
  - statement `n:=n+1` may cause `a[n]` to go out of bounds
- need a more sophisticated method that avoids array bound problems

Write this instead, which dynamically resizes if the array is full


```
method Push(x: Data)                  // same as before
ensures shadow == old(shadow) + [x]; // same as before
{
   if n==buf.Length                   If n==buf.Length && m==0 then the array is full!
    {
       var b:= buf; // this gives b the old size
       if m==0 { b:= new Data[2*buf.Length]; } // this doubles the size
       forall (i | 0<=i<n-m) { b[i]:= buf[m+i]; } // copy leftmost
       buf, m, n:= b, 0, n-m; // quack contents now starts at m=0, length n-m
    }
    buf[n], n:= x, n+1;        // now we definitely have room in the array
    shadow:= shadow + [x];     // same as before
}
```

Lecture 12

Final slide

Two kinds of verification of high-level data structures:

- Program verification on the data structure code:
  - Treats the data structure operations as program code
  - Verify operations using pre- and post-conditions
  - Valid() defines 'correct' actual data
    - Valid() is maintained by every operation
  - We will see two examples of this: Hardstack and Cyclestore
- Ghost verification on a ghost data structure, typically called *shadow*
  - Verify operations using pre- and post-conditions on the shadow
  - As before, Valid() defines 'correct' actual data
    - Valid() also specifies the relationship between shadow and the actual data
  - We will see two examples of this: Quack and Queue

Example of using our Quack

Note that the Quack will dynamically resize based on the state of the quack when pushing. If its full it will dynamically resize and make a new Quack twice the size with all the old data.

There are two different versions of "being full"

- When n is at the limit, but m is not 0
  - Array is not really full, so all we have to do is reshift everything back down from 0 to n-m
- When n is at the limit and m is 0
  - This case is when it is really fully, when all indices are taken up, we need to create our new array of twice size

```
if n==buf.Length                    If n==buf.Length && m==0 then the array is full!
 {
    var b:= new Data[buf.Length];           // temporary buffer, same size
    if m==0 { b:= new Data[2*buf.Length]; } // double size of temporary
    forall (i | 0<=i<n-m) { b[i]:= buf[m+i]; } // copy m..n to 0..n-m
    buf, m, n:= b, 0, n-m;           // temporary is now buf, reset m and n
 }

method Main() // tested on 1.9.7
{   var q:= new Quack<char>(10);
    var l: char;
    q.Push('r'); q.Push('s'); q.Push('k'); q.Push('o'); q.Push('w');
    l:= q.Pop(); assert l=='w'; print l;
    q.HeadTail();  // swap head/tail: an assignment exercise
    l:= q.Qop(); assert l=='o'; print l;
    l:= q.Pop(); assert l=='r'; print l;
    q.HeadTail();
    l:= q.Qop(); assert l=='k'; print l;
    l:= q.Qop(); assert l=='s'; print l;
    var e: bool:= q.Empty(); assert e;
    if e {print "\nqueue empty\n";} else {print "\nqueue not empty\n";}
}
```

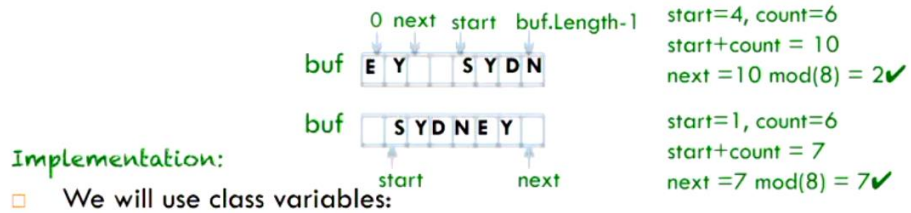
Will print, "works" from all those pops and qops

Cyclical Queue (HARDEST CONTENT IN COURSE)

- Queue only has 1 index to consider, the start
- Queue also needs to keep track of its size
- When we reach the end we wrap around to the start
    - The queue will cycle around the array if we reach the end and restart from the beginning as if it were never fill
        - SYD → full
            - Prints SYD
        - NEY SYD → full
            - Prints SYDNEY
        - NEY ROCKS SYD → full
            - Prints to Sydney Rocks
- We still use the shadow to mirror the actual sequence of data

Implementation strategy:

- pop from the head, at index *start*
- push onto the tail, at index *next = (start + count) mod (buf.Length)*



Implementation:

- We will use class variables:

```
buf:array<Data>; start:nat; count:nat;
```

- The actual queue:
    - Could be split in two parts, where 2nd part precedes 1st part
    - Or could be contiguous

Notice how now the array has 2 sections to consider, if and only if start > next (meaning we wrapped around)

For this queue we will use a ghost sequence to shadow the real data

Our invariant needs to

- Place limits on start and count
- The queue may or may not be contiguous
    - If the queue fits into the array, (start + count <= array length, then the shadow is just the array from start → start + count)
    - If the queue does not fit into the array (start + count > array length, then the shadow will be the two slices glued together)
        - First slice
            - Array from start to end of array
        - Second slice
            - Array from 0 to start + count – length of array

```
class {:autocontracts} CycleQueue<Data>
{
 var buf: array<Data>; // the cycle buffer
 var start: nat;       // index of the first element of type Data
 var count: nat;       // number of elements

 ghost var shadow: seq<Data>;  // shadow the real queue

 predicate Valid()
 { buf != null && buf.Length > 0 &&
   count <= buf.Length &&
   start < buf.Length &&
   if start+count <= buf.Length
   then shadow == buf[start..start+count]      single part
   else shadow == buf[start..] + buf[..start+count-buf.Length]
 }
                2nd part                1st part
```

```
 constructor (size: nat)
 requires size>0          // must be non-zero buffer
 ensures buf.Length==size
 ensures shadow==[]       // shadow starts clean
 { buf := new Data[size];
   start, count := 0, 0;
   shadow := [];
 }

 method Head() returns (x:Data)
 requires buf != null   // 1.9.7
 requires shadow != []  // can't be empty
 ensures x==shadow[0]   // x is also the head of shadow
 { x := buf[start]; }   // x is the buf element at index start
```

Note head is just a way to peek into the queue

```
method Pop() returns (x: Data)
requires buf != null
requires shadow != []            // must be something there
ensures x==old(shadow)[0]        // return the bottom
ensures shadow==old(shadow)[1..] // remove the bottom
{
   x := buf[start];
   if start==buf.Length-1 {start:=0;} else {start:=start+1;}
   count := count - 1;
   shadow := shadow[1..];
}
              The ghost Pop
```

Note the if statement is where the cycling occurs in our array, if we pop at the end, reset

More complicated push

```
method Push(x: Data)
ensures shadow==old(shadow) + [x]
ensures if old(count==buf.Length) then buf.Length==2*old(buf.Length)
                                  else buf.Length==old(buf.Length)
{
  if count==buf.Length {                    // need more room
    var double := buf.Length;
    var d := new Data[buf.Length + double];  // create a bigger buffer
    forall i| 0<=i<buf.Length {
      d[if i<start then i else i + double] := buf[i];
    }                                        // elements >= start move up to 2nd half
    buf, start := d, start + double;
  }
  var next := start + count;
  if next<buf.Length { buf[next] := x;} else { buf[next - buf.Length] := x; }
  count := count + 1;
  shadow := shadow + [x];
} } end of CycleQueue
```

The ghost push

The real push operation

To maintain the cyclical nature notice the if inside the executable for all.

All elements that are from the start to the end will get shifted to the second half

All new elements from overflow will be pushed from the initial index

```
method Test()
{
   var a:int, b:int, c:int, d:int, e:int, f:int;
   var cq := new CycleQueue<int>(2);
   cq.Push(a);
   cq.Push(b);
   cq.Push(c);
   cq.Push(d);
   cq.Push(e);
   var x := cq.Pop(); assert x==a;
        x := cq.Pop(); assert x==b;
        x := cq.Pop(); assert x==c;
        x := cq.Pop(); assert x==d;
        x := cq.Pop(); assert x==e;
   cq.Push(f);
        x := cq.Pop(); assert x==f;
}
Dafny program verifier finished with 7 verified, 0 errors
```

Look! No hands!
I'm testing without defining
the values of the test data!

See how it dynamically resized even though the array was initially sized 2.

Below will highlight the cyclical push

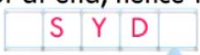
In essence, doubling the array size creates room in the middle of the array in which to push data elements

											start	count	'next'
							[	.	.	]	0	0	0
Push a							[	**a**	.	]	0	1	1
Push b							[	**a**	b	]	0	2	0
Push c					[	c	.	**a**	b	]	2	3	1
Push d					[	c	d	**a**	b	]	2	4	2
Push e	[	c	d	e	.	.	.	**a**	b	]	6	5	3
Pop→a	[	c	d	e	.	.	.	a	**b**	]	7	4	3
Pop→b	[	**c**	d	e	.	.	.	a	b	]	0	3	3
Pop→c	[	c	**d**	e	.	.	.	a	b	]	1	2	3
Pop→d	[	c	d	**e**	.	.	.	a	b	]	2	1	3
Pop→e	[	c	d	e	.	.	.	a	b	]	3	0	3
Push f	[	c	d	e	**f**	.	.	a	b	]	3	1	4
Pop→f	[	c	d	e	f	.	.	a	b	]	4	0	4

41

```
method Push(x: Data)
ensures shadow==old(shadow) + [x]
ensures if old(count==buf.Length)
        then buf.Length==2*old(buf.Length)
        else buf.Length==old(buf.Length)
{   Needed because of timeout problems, not logically
 if count==buf.Length { // completely full
 var double := buf.Length;
 var d := new Data[buf.Length+double];
 forall i| 0<=i<buf.Length
  {d[if i<start then i else i+double] := buf[i];}
 buf, start := d, start+double;
 } // if statement
 var next := start+count;
 if next<buf.Length { buf[next]:=x;}
              else { buf[next-buf.Length]:=x; }
 count := count + 1;
 shadow := shadow + [x]; } // method
```

- Not at end, hence there's room



next := start+count - Push N

- Reached end, but there's still room

Push E - next := start+count - buf.Length

- Completely full

Push Y

next := start+double + count - buf.Length

Law of Explosion

- Under a contradiction we are able to say anything is true
    - If the sky is orange
        - Then pigs can fly
            - THIS IS WILL MEAN PIGS CAN FLY
                - HOWEVER THIS IS POINTLESS SINCE THE SKY IS NEVER ORANGE

1	a	Premise 1
2	!a	Premise 2
3	pigsfly	Aim: to prove this
4	a || pigsfly	Disjunction introduction on 1
5	pigsfly	Disjunction syllogism using 4 and 2

```
Disjunction introduction
p ==> p||q

Disjunction syllogism
(p||q) && !p ==> q
```

From a contradiction we can prove anything, under any false assumption anything is true

If 1 == 2, then 3 == 4 is true.

Assumptions (this is terrible never use this trash)

```
method AssumeTrueWhatIsFalse() {
  assume 1==2;  // is very false
  assert 3==4;
}
```

Seems contradictory

```
Dafny program verifier finished with 1 verified, 0 errors
Error: an assume statement cannot be compiled
```

Note under assumptions we cannot compile, we are almost forcing the program to always return 0 errors with this.

If we assume something that is true, it will behave correctly

```
method AssumeTrueWhatIsTrue() {
  assume 1==1;  // is true
  assert 3==4;
}
```

This will give us an assertion violation, since the assumption is true and can be provably true.

What happens in the following case?

```
method AssumeFalseWhatIsFalse() {
  assume !(1==2);
  assert 3==4;
}
```

Assumptions are dangerous, avoid them at all costs. Its like tightening your pre-condition or weakening your post condition to force your program to work, we want it to work under no assumptions

Power of 2

A great specification but horrible implementation of the power of 2 would be the following

```
function funpower(n: nat): nat
{ if (n==0) then 1 else 2*funpower(n-1) }
```

Similar to the example of Fibonacci. When the n > 100 we have > 100 stack frames, very bad

So we use this as our specification since it is mathematically defined, and then we will implement it iteratively

```
method SlowPower(n: nat) returns (p: int) // returns 2^n
ensures p==funpower(n)
{
  var i := 0;
  p := 1;
  while i<n // compute nth power of 2 iteratively
  invariant 0<=i<=n
  invariant p==funpower(i)
  {
     p := p*2;
     i := i+1;
  }
}
```

Works but is slow. Loops n times, very similar example to Fibonacci.

We can make a fast power which uses indices as a binary of 2^n, and will compute it in log(n) time

2^{1024} requires 1023 multiplications

- that's a lot of work if you need it often

We can use the fact that:

- $2^{2n} == 2^n * 2^n$
- $2^{2n+1} == 2^{2n} * 2$

2^{1024} just needs 9 multiplications

```
a := 2 * 2;
a := a * a;   == 2^2 * 2^2
a := a * a;   == 2^4 * 2^4
a := a * a;   == 2^8 * 2^8
a := a * a;   == 2^16 * 2^16
a := a * a;   == 2^32 * 2^32
a := a * a;   == 2^64 * 2^64
a := a * a;   == 2^128 * 2^128
a := a * a;   == 2^512 * 2^512
```

2^9 needs 4 multiplications

```
a:=2 * 2;
a:=a * a;   == 2^2 * 2^2
a:=a * a;   == 2^4 * 2^4
a:=a * 2;   == 2^8 * 2
```

The number of multiplications is approx. log of n, where n is the exponent (e.g. $\log_2 1024 = 10$, $\log_2 9 \approx 3$)

(he is missing 2^256 which is 2^128 * 2^128)

This is using the law of indices that a^(b+c) = a^b * a^c.

'Fast' way of computing exponentials does log(n) multiplications

- It's recursive, but different to the spec!

Also called the Russian peasant algorithm. One of the oldest algorithms in existence

```
method FastPower(n: nat) returns (p: int)
{
    if n==0 {
        p := 1;
    } else if n%2 == 0 {
        p := FastPower(n/2); // if p is even then
        p := p*p;            // FastPower(n):=FastPower(n/2)*FastPower(n/2)
    } else {
        p := FastPower(n-1);
        p := 2*p;
    }
}
```

- We can verify this the same as we did for SlowPower()

However if we give it the following post condition

```
ensures p==funpower(n)
```

Dafny fails to verify it because it doesn't know the following law:

$$2^{a+b} == 2^a \times 2^b$$

- in our case $2^n == 2^{n/2} \times 2^{n/2}$

Dafny will cry since it cannot make the jump from the method instructions to the specifications, it doesn't know the laws of indices, it only knows very basic axioms.

SO HOW DO WE FIX THIS

Lemmas

- A ghost method
- Has a pre condition and post condition
- Gives us pre condition implies post condition

```
lemma lawofexponents(n: nat)
requires n%2==0 // so n must be even
ensures funpower(n) == funpower(n/2) * funpower(n/2) // law here
{
   if (n!=0) {lawofexponents(n-2);} // how to prove it here
}  Dafny proves this lemma using induction.
```

Dafny will use induction to prove our lemma and does this before any other verification.

Look at this example to see how the lemma works

```
lawofexponents(8)
requires 8%2==0
ensures funpower(8) == funpower(4)*funpower(4)
lawofexponents(8-2)
▫ lawofexponents(6)
   requires 6%2==0
   ensures funpower(6) == funpower(3)*funpower(3)
   lawofexponents(6-2)
   ▪ lawofexponents(4)
     requires 4%2==0
     ensures funpower(4) == funpower(2)*funpower(2)
     lawofexponents(4-2)
      ▪ lawofexponents(2)
        requires 2%2==0
        ensures funpower(2) == funpower(1)*funpower(1)
        lawofexponents(2-2)
         ▪ lawofexponents(0)
           return
```

It will go down to the very base case and check if the base case is true, then it will use that result to prove the next case is true which is built from the base case, then it will do that for the next one built off the previous one and so on, inductively.

- No change in state
- No side effects
- No returns

```
lemma SomeName(x:T) // or 'ghost method SomeName(x:T)'
requires P(x)
ensures Q(x)
{ body }
```

- If the method verifies then we must have *forall x:T:: P(x) ==> Q(x)*
- Useful to partition (or structure) a proof:
  - Dafny will prove lemmas first
  - Dafny will then prove the rest of the program

Now we can call our lemma in our function anywhere, and dafny will first do lemma verification and LEARN the result of the lemma to be used in method proof

```
method FastPower(n: nat) returns (p: int)
ensures p==funpower(n)
{
    if n==0 {
       p := 1;
    } else if n%2 == 0 {
       lawofexponents(n);  <-- this lemma helps Dafny prove the post-condition
       p := FastPower(n/2);
       p := p*p;
    } else {
       p := FastPower(n-1);
       p := 2*p;
    }
}
Dafny program verifier finished with 2 verified, 0 errors.  yey!
```

If we keep getting errors even though we are right, try using a lemma to prove something that is provable.

Searching

We want to create our search method to look for a key in an array of integers.

- If the key is in the array, return the first index i
- If the key is not in the array, return -1

Now that we know the specification, we can create our search.

Linear Search

- No requirements, can work on any array

```
the spec
method Find(a: array<int>, key: int) returns(i: int)
ensures 0<=i ==> i<a.Length && a[i]==key
ensures i<0  ==> forall k:: 0<=k<a.Length ==> a[k]!=key
{
  i:=0;
  while i<a.Length
  invariant 0<=i<=a.Length
  invariant forall k:: 0<=k<i ==> a[k]!=key
  {
      if a[i] == key { return; }
      i:=i+1;
  }
  i:=-1;
}
```

This invariant is strong enough for Dafny to prove the spec will never be violated

Very simple

Binary Search

- Our array MUST be sorted for this to work

```
method Find (a:array<int>, key:int) returns (index:int)
// spec required here
{
   var low, high := 0, a.Length;
   while low < high // terminate when low==high
   {
      var mid := (low+high)/2;
      if key > a[mid]
         {low := mid + 1;}
      else if key < a[mid]
         {high := mid;}
      else
         {return mid;} // found it
   }
   return -1; // low==high
}
```

However, we are missing our two specifications

1. We need to require sorted
2. We need our post condition to define the return result
3. We need an invariant to link the loop to our post condition

Lecture 13

If we assume falsehood → verification is pointless, if we assume 1 == 2, whatever we go proving with that assumption is rubbish because 1 is never == 2. If you assume something true, it will just ignore the assumption and continue as normal.

There are times however where things aren't as black as white as assuming 1 == 2, sometimes we can assume an array is sorted, or assume some values of an array. Note its still dangerous but sometimes can be useful (not recommended though).

Dafny can use assertions to verify in some cases, but it wont compile since it has assume.

```
method test() {
  var a:= new int[4];                                  // array can be anything
  assume forall i,j:: 0<=i<j<a.Length ==> a[i]<=a[j]; // assume ordered
  assume a[0]==a[1]==a[2]==a[3]==0;                    // assume all the same
  assume a[0]==1;    // CLEARLY A WRONG ASSUMPTION!!!
  assert |a[..]| == 10;
}
Dafny program verifier finished with 1 verified, 0 errors   No errors??
Error: an assume statement cannot be compiled
Error: an assume statement cannot be compiled
Error: an assume statement cannot be compiled
```

The false assertion is not detected.

So, making a wrong assumption, makes verification meaningless.

(For development purposes, 'assumes' can be useful, but never in submitted code.)

We could add assumes to check, but do not use them in submission.

Binary Search

We can use the same post condition and add the pre-condition that the array must be sorted. But now we need an invariant for the new loop.

- Bound the low and high indexes
- Finding the key, so our invariant is that all values outside the range of low and high are NOT the key

1. The while-loop condition is `low<high`

   The invariant must put limits on the indices low and high

   `invariant 0<=low<=high<=a.Length`

2. The invariant must also 'support' the forall clause in the post-condition

   `forall k:: 0<=k<a.Length ==> a[k]!=key`

   ... which is the same for linear and binary search.

   Linear search's invariant stated the key was not yet found at index i

   `invariant forall k:: 0<=k<i ==> a[k]!=key // linear search`

   For binary search, the loop terminates when low equals high

   `invariant forall k:: 0<=k<a.Length && !(low<=k<high) ==> a[k]!=key //binary`

   - that is, there is no key outside the range low..high-1

Need to think, what do we know, once you understand what you're doing it all makes sense. In a binary search the ONLY thing we know for sure is that every time we half, we know whatever we TOOK OFF is not the key.

You need to ask yourself, what is your loop doing, the invariant is just describing what your loop does.

```
method Find(a: array<int>, key: int) returns(i: int)
requires a!=null && allsorted(a)
ensures 0<=i ==> i<a.Length && a[i]==key
ensures i<0  ==> forall k :: 0<=k<a.Length ==> a[k]!=key
{
   var low, high := 0, a.Length;
   while low < high
   invariant 0<=low<=high<=a.Length
   invariant forall k:: 0<=k<a.Length && !(low<=k<high) ==> a[k]!=key
   {
      var mid := (low+high)/2;
      if key > a[mid]
           {low := mid + 1;}
      else if key < a[mid]
           {high := mid;}
      else {return mid;}
   }
   return -1;
}
```

if low==high this term disappears leaving the postcondition

```
Dafny program verifier finished with 3 verified, 0 errors
```

Insertion Sort

We need to be careful when we sort, if our post condition is the array is sorted, we can simply wipe the initial array and store in 1, 2, 3, 4 etc.

Consider the following program

```
predicate Sorted(a:array<int>)
reads a;
{ forall j,k:: 0<=j<k<a.Length ==> a[j]<=a[k] }

method MysterySort(a:array<int>)
ensures Sorted(a);
modifies a;
{ var i:=0;
  while i<a.Length
  invariant 0<=i<=a.Length;
  invariant forall j::0<=j<i ==> a[j]==0;
  { a[i]:=0; i:=i+1; } // corruption
}

method Verify()
{
   var a:= new int[][3,5,1,4,6,2];
   MysterySort(a);
   assert Sorted(a);
}
```

The array 'a' is corrupted, but sorted

We want to make sure that the multiset of the previous array and the multiset of the sorted array are identical to check the values are the same!

```
method SomeSort(a:array<int>)
ensures Sorted(a);
ensures multiset(a[..]) == multiset(old(a[..]));
modifies a;
```

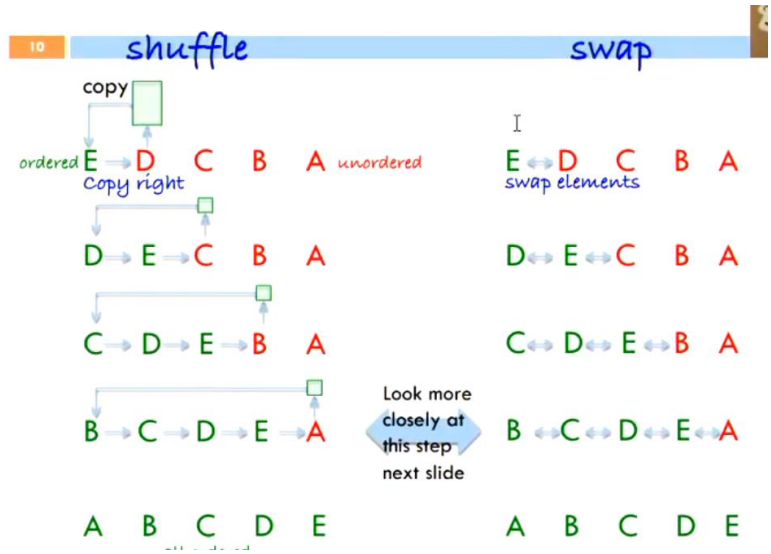
Or

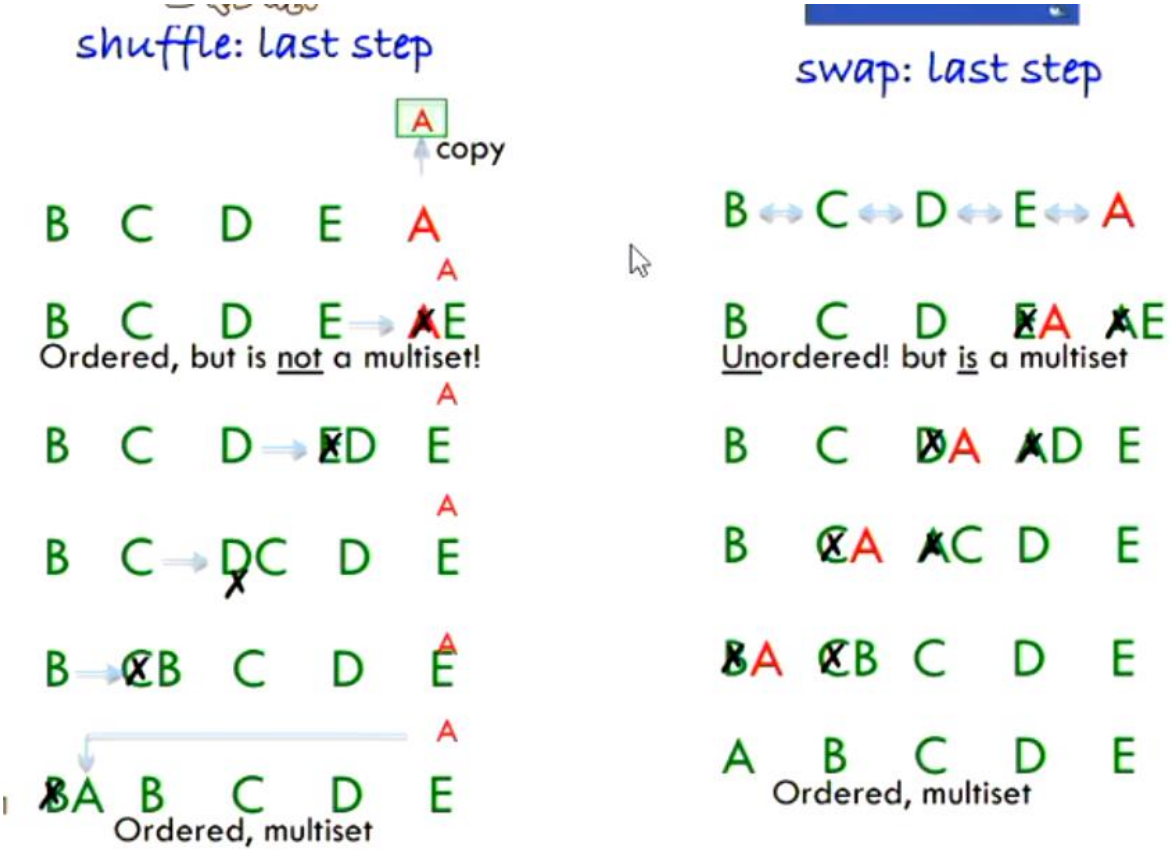
```
method SomeSort(a:array<int>) returns (b:array<int>)
ensures Sorted(b);
ensures multiset(b[..]) == multiset(a[..]);
```

Insertion Sort Dafny

Two methods

- Shuffle
- Swap





Notice in the shuffle approach we actually corrupt to sift values down, we get moments in time where we lose the actual original array values while we shift values up, however it will always maintain the sorted.

In the swap, we don't swap and through every step, the values are the same as the original, but its not ordered till the end.

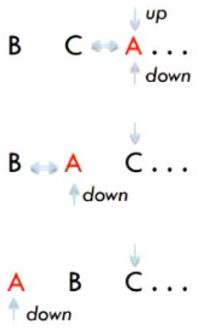
In the end they are both ordered multiset.

This will form the basis of our invariant.

For the shuffle we can say that everything to the left of the value will be sorted.

Swap approach

```
var up:=1;
while (up<a.Length)
{
   var down := up;
   while (down >= 1 && a[down-1] > a[down])
   {
      a[down-1], a[down] := a[down], a[down-1];
      down:=down-1;
   }
   up:=up+1;
}
```



Now we need to write specification

```
method InsertionSortSwap(a: array<int>) // an 'in situ' swap
requires a.Length>1
ensures Sorted(a, 0, a.Length);
ensures multiset(a[..]) == multiset(old(a[..]));
modifies a;

{
  var up:=1;                    // the first unordered element is at index 1
  while (up<a.Length)           // increment up to the last index in array
  invariant 1 <= up <= a.Length;              // the usual limits
  invariant Sorted(a, 0, up);                 // everything <up is done
  invariant multiset(a[..]) == multiset(old(a[..])); // notice, whole array!
  { .
    .          Why is the whole array in the multisets?
    .
```

Note we make sure the WHOLE Array is in the multiset because with the swap approach it says that the multiset is the same throughout, we don't corrupt at any point.

It would be wrong to use `multiset(a[..up]) == multiset(old(a[..up]))`

- **violations:** loop not maintained, postcondition may not hold

In the shuffle approach we only know the multiset up until up is the same, but its sorted the whole way through before up.

Now we need an invariant for the inner loop since both loops do different things

The inner loop is very simple:

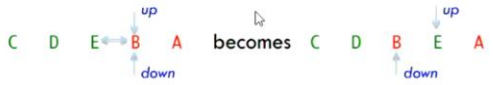
- *down* clearly goes from *up* to 0
- multiset is the same as the outer loop (multiset of whole does not change)

```
var down := up;
while (down >= 1 && a[down-1] > a[down])
invariant 0<=down<=up;
invariant multiset(a[..]) == multiset(old(a[..]))
{
   a[down-1], a[down] := a[down], a[down-1];
   down:=down-1;
}
```

So simple, we just bound our down, and maintains the same property that the multiset doesn't change. These invariants say what is true in every iteration, it describes what happens before and after each iteration.

We are missing one invariant in the inner loop, however. We need to say what is sorted after each iteration. After each iteration we can say anything between down and up is sorted and this is the final step of the puzzle.

If you ignore the 'rogue' element *down* then the array is sorted from *0* to *up*



But it's not quite that simple:

- from *down* to *up* the array is sorted (we swap to make it that way)
  - the problem is to the left of *down*
- in fact, all pairs `(i,j)` for `j==down` are <u>not sorted</u>
- so we must exclude these in the invariant

```
invariant forall i,j:: (0<=i<j<=up && j!=down)==>a[i]<=a[j]
```


```
method InsertionSortSwap(a: array<int>)
requires a.Length>1
ensures Sorted(a, 0, a.Length);
ensures multiset(a[..]) == multiset(old(a[..]));
modifies a;
{
  var up:=1;
  while (up < a.Length)
  invariant 1 <= up <= a.Length;
  invariant Sorted(a, 0, up);
  invariant multiset(a[..]) == multiset(old(a[..]));
  {
    var down := up;              // the next unordered element
    while (down >= 1 && a[down-1] > a[down])
    invariant 0 <= down <= up;
    invariant forall i,j:: (0<=i<j<=up && j!=down)==>a[i]<=a[j];
    invariant multiset(a[..]) == multiset(old(a[..]));
    {
      a[down-1], a[down] := a[down], a[down-1];
      down:=down-1;
    }
    up:=up+1;
  }
}
Dafny program verifier finished with 3 verified, 0 errors
```

Lecture 14

Upfront

Partially ordering an array so all 0's are up at the front

- Must be in situ
- No other data structures but array
- The order of all the non-zero elements are allowed to change
    - Only care for 0's upfront

The post condition has to say that all the values between 0 and the non-zero range are 0. It should also make sure that the multiset remains the same

```
method Upfront(a: array<int>) returns (nonz: int)
ensures 0 <= nonz <= a.Length
ensures forall i:: 0    <=i <nonz     ==> a[i]==0
ensures forall i:: nonz <=i <a.Length ==> a[i]!=0
ensures multiset(a[..]) == multiset(old(a[..]))
modifies a;
{
    nonz := 0;
    var next := 0;
    while (next != a.Length)
    invariant 0 <= nonz<=next<= a.Length
    invariant forall i:: 0   <=i <nonz ==> a[i]==0
    invariant forall i:: nonz<=i <next ==> a[i]!=0
    invariant multiset(a[..]) == multiset(old(a[..]))
    {
       if (a[next]==0) {
          a[next], a[nonz] := a[nonz], a[next];
          nonz:=nonz+1;
       }
       next:=next+1;
    }
}
```

```
method Main()
{
  var a: array<int> := new int[8];
  a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7] := 2, 0, 1, 5, 3, 0, 4, 0;
  assert a[..] == [2, 0, 1, 5, 3, 0, 4, 0];
  ghost var ms := multiset(a[..]); // ms is used just to verify (not printed)

  print a[..], '\n';
  var n := Upfront(a);
  assert forall i:: 0<=i<a.Length ==> if i<n then a[i]==0 else a[i]!=0;
  assert ms == multiset(a[..]);
  print "number of zeros = ", n, '\n';
  print a[..], '\n';
}
```

results in ...

```
[2, 0, 1, 5, 3, 0, 4, 0]
number of zeros = 3
[0, 0, 0, 5, 3, 2, 4, 1]
```

Takes about 3 secs using 1.9.7 on CSE's 'weber'
If we take the testcase out (in green), it takes about 2 secs.

Dutch Flag

This is an extension on upfront.

Sorting the colors of the Dutch flag, linear time

The best performing strategy:

- store the 'unsorted balls' in an array
- partition the array into 4 sections
  - Red section, initially empty
  - White section, initially empty
  - Unsorted section, initially the whole array
  - Blue section, initially empty
- use 3 pointers, indicating
  - white: Start of white balls (initially 0)
  - next: Start of unsorted balls (initially 0)
  - blue: Start of blue balls (initially array.Length)

Pseudo code

- If next==WHITE then increment next
- If next==BLUE then decrement blue & swap A[next] and A[blue]
- If next==RED then swap A[next] and A[white] & increment white & increment next
- terminate when next == blue

This is a linear, in-situ sorting algorithm amazing.

Here is the verification required

```
ensures 0 <= white<=blue <= flag.Length
ensures forall i:: 0     <=i <white       ==> flag[i]==RED
ensures forall i:: white <=i <blue        ==> flag[i]==WHITE
ensures forall i:: blue  <=i <flag.Length ==> flag[i]==BLUE
```

```
invariant 0 <= white<=next<=blue <= flag.Length
invariant forall i:: 0    <=i<white       ==> flag[i]==RED
invariant forall i:: white<=i<next        ==> flag[i]==WHITE
invariant forall i:: blue <=i<flag.Length ==> flag[i]==BLUE
```

unsorted part of the array

Note you can write the ensures differently by ensuring the result set is sorted

We also need to ensure that the multiset of our result remains the same

```
datatype Colour = RED | WHITE | BLUE

method FlagSort(flag: array<Colour>) returns (white:int, blue:int)

   ensures forall i:: 0     <=i <white       ==> flag[i]==RED
   ensures forall i:: white <=i <blue        ==> flag[i]==WHITE
   ensures forall i:: blue  <=i <flag.Length ==> flag[i]==BLUE
   ensures multiset(flag[..])== multiset(old(flag[..]))   multiset
   modifies flag;
   {
       var next := 0;
       white := 0;
       blue := flag.Length;    // colours between next and blue unsorted
       while (next != blue)    // if next==blue, no colours left to sort
       invariant 0 <= white<=next<=blue <= flag.Length        indexes
       invariant forall i:: 0     <=i<white       ==> flag[i]==RED
       invariant forall i:: white<=i<next         ==> flag[i]==WHITE
       invariant forall i:: blue <=i<flag.Length ==> flag[i]==BLUE
       invariant multiset(flag[..])== multiset(old(flag[..])) multiset
       {
               match (flag[next])
               {
               case RED   => flag[next], flag[white] := flag[white], flag[next];
                             next:=next+1;
                             white:=white+1;
               case WHITE => next:=next+1;
               case BLUE  => blue:=blue-1;
                             flag[next], flag[blue] := flag[blue], flag[next];
               }
       } }
```

Verifying FlagSort

```
method Main()
{
  var flag: array<Colour> := new Colour[6];
  flag[0], flag[1], flag[2], flag[3], flag[4], flag[5]:= BLUE, RED, WHITE, BLUE, RED, WHITE;
  var ms  := multiset(flag[..]);
  var a, b := FlagSort(flag);
  assert Reds(flag, a) && Whites(flag, a, b) && Blues(flag, b);
  assert ms == multiset(flag[..]);
  print a, ' ', b, ' ', flag[..], '\n';
  flag := new Colour[0];
  ms := multiset(flag[..]);
  a, b := FlagSort(flag);
  assert Reds(flag, a) && Whites(flag, a, b) && Blues(flag, b);
  assert ms == multiset(flag[..]);
  print a, ' ', b, ' ', flag[..], '\n';
}
```

Verification takes about 4 secs (any size array) using 1.9.7 on CSE's 'weber'.

Execution:

```
2 4 [colour.RED, colour.RED, colour.WHITE, colour.WHITE, colour.BLUE, colour.BLUE]
0 0 []
```

Partitioning

Portioning an array

- On left: elements smaller than pivot
- On right: elements bigger than pivot

For example

- 9 5 4 2 8 7 3 1 5 choose the last element to be the 'pivot'
- 1 3 4 2 5 7 5 9 8 notice numbers are partitioned, not sorted

- Must be in situ
- No other data structures can be used

This is the fundamental building block to quicksort which is the most popular sort in the world. It is very fast in a majority of cases, o(nlogn) on average case.

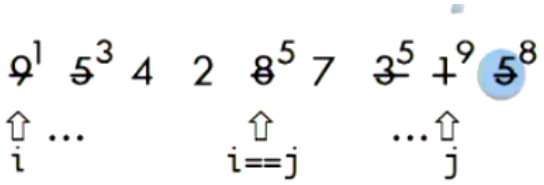
Pivoting

Pick the last element as the pivot

Introduce two new variables, low and high

- Low starts at 0, and high starts at the element BEFORE the pivot

I. Increment i until a[i]>=pivot
II. Decrement j until a[j]<pivot
III. Swap a[i] and a[j]
IV. Repeat I. to III. until i==j
V. Swap a[j] and pivot



We have three while loops now

- Outer loop for I and j to meet each other
    - Inner loop 1: for I to move to the right across the correctly positioned numbers
    - Inner loop 2: for j to move to the left across correctly positioned numbers
- Three invariants needed, but the inner loops are very simple
- The portioning strategy is linear since I and j each move across the array once

```
method Partition(a: array<int>) returns (pi: int) // pivot index
{
    var last := a.Length-1;   // a[last] is the pivot
    var i := 0;
    var j := last-1;
    while (i < j)
    {
        while (i<j && a[i]<a[last])
        {i:=i+1;}  // dances to the right until an upper element is found

        while (i<j && a[j]>=a[last])
        {j:=j-1;}  // dances to the left until a lower element is found

        if (i<j) {
           a[i], a[j] := a[j], a[i];
           i:=i+1;
        }
    }
    a[i], a[last] := a[last], a[i]; // put the pivot where it belongs
    return i;
}
        At this point, either i==j and we're finished, or
15      a[i]≥a[last]>a[j] and we need to swap and iterate again
```

Pi, the pivot is returned by the method, and the goal is to put the initial pivot (last element) in the correct position in the array. Everything to the left of the pivot is smaller than the pivot, everything to the right of the pivot is bigger than the pivot

- everything to the left of i is <a[pi]

```
predicate Lower (a:array<int>, i:int, pi:int) reads a
requires 0<=i<a.Length && 0<=pi<a.Length
{ forall k:: 0 <= k < i ==> a[k]<a[pi] }
```

- everything to the right of j is ≥a[pi]

```
predicate Upper (a:array<int>, j:int, pi:int) reads a
requires 0<=j<a.Length && 0<=pi<a.Length
{ forall k:: j < k < a.Length-1 ==> a[k]>=a[pi] }
```

Our invariant is expressed using these predicate definitions

```
method Partition(a: array<int>) returns (pi: int)
requires a.Length>1
ensures 0<=i<a.Length
ensures Lower (a, pi, pi)   // everything below index pi is < a[pi]
ensures Upper (a, pi, pi)   // everything above index pi is >= a[pi]
ensures multiset(a[..]) == multiset(old(a[..]))
modifies a;
{ . . .
  Partition body
  . . .
}
```

Note the second pi is for the value of the pivot

```
var last := a.Length-1;   a[last] is the pivot
var i := 0;
var j := last;            i and j start at the opposite ends
while (i < j)             as long as i and j don't 'meet'
invariant 0<=i<=j<=last
invariant Lower(a, i, last) // everything below index i is <a[last]
invariant Upper(a, j, last) // everything above index j is >=a[last]
invariant multiset(a[..]) == multiset(old(a[..]))  no change to the multiset
{  .
   .
```


```
method Partition(a: array<int>) returns (pi: int)
requires . . . // slide 17
ensures . . .  // slide 17
modifies a
{
    var last := a.Length-1;   // a[last] is the pivot
    var i := 0;
    var j := last;
    while (i < j)
    invariant 0<=i<=j<=last
    invariant Lower(a, i, last) && Upper(a, j, last)
    invariant multiset(a[..]) == multiset(old(a[..]))
    {
        while (i<j && a[i]<a[last])
        invariant 0<=i<=j
        invariant Lower(a, i, last) // everything below i is <a[last]
        {i:=i+1;}
        while (i<j && a[j]>=a[last])
        invariant i<=j<=last
        invariant Upper(a, j, last) // everything above j is >=a[last]
        {j:=j-1;}
        assert i<=j;
        if (i<j) { // swap because a[i]>=a[last]>a[j]
           a[i], a[j] := a[j], a[i];
           i:=i+1;
        }
    }
    assert i==j;  // we've met!
    a[i], a[last] := a[last], a[i]; // the pivot is in the right spot
    return i;
}
```

Partition testing ...


```
method Main()
{
  var a: array<int> := new int[12];
  a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10], a[11]
      := 9,5,6,4,6,2,8,7,3,1,10,6;
  ghost var ms := multiset(a[..]); // keep it invisible
  print a[..], '\n';
  var n := Partition(a);
  assert Lower(a, n, n) && Upper(a, n, n);
  assert ms == multiset(a[..]);
  print "pivot index is ", n, '\n', a[..], '\n';
}
```

everything before n is < a[n] && everything after n is ≥ a[n]

```
Dafny program verifier finished with 6 verified, 0 errors
Program compiled successfully
Running...

[9, 5, 6, 4, 6, 2, 8, 7, 3, 1, 10, 6]
pivot index is 5
[1, 5, 3, 4, 2, 6, 8, 7, 6, 9, 10, 6]
```

the last element is the pivot

verifies in about 2 secs, with or without the testcase

the pivot gets moved here

What is the Partition() of the array [9,8,7,6,5,4,3,2,1]?
What are Lower and Upper?

Last Lecture

Quicksort

- Very fast, a lot faster than insertion sort
    - Average case O(nlogn)
    - Worst case O(n^2)
- partitioning is linear
- each partition ill half the length of the array

the difference in performance is that, if sorting 1000 items takes 1 micro second, then sorting 1 billion items with

- insertion sort
    - 32 years
- Quick sort
    - 20 seconds

Dancing pair partition

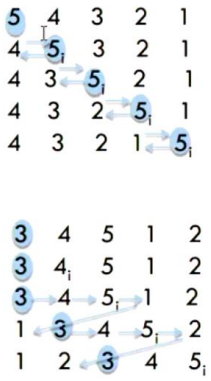
Partition body complete


```
{
  var last := a.Length-1;   // a[last] is the pivot
  i := 0;
  var j := last;
  while (i < j)
  invariant 0<=i<=j<=last
  invariant Lower(a, i, last)
  invariant Upper(a, j, last)
  invariant multiset(a[..]) == multiset(old(a[..]))
  {
      while (i<j && a[i]<a[last])
      invariant 0<=i<=j
      invariant Lower(a, i, last) // everything below index i is < a[last]
      {i:=i+1;}

      while (i<j && a[j]>=a[last])
      invariant i<=j<=last
      invariant Upper(a, j, last) // everything above index j is >= a[last]
      {j:=j-1;}

      if (i<j) {
         a[i], a[j] := a[j], a[i];
         assert a[i] < a[last] <= a[j]; // i and j are in correct order
         i:=i+1;
      }
  }
  a[i], a[last] := a[last], a[i]; // put the pivot where it belongs
}
```

Quicksort will call the partitioning algorithm and put the pivot in the correct position. All quicksort does is just partition the array recursively until the array becomes sorted. It is more of a partition algorithm.

```
method QuickSort (a:array<int>, low:int, high:int)
requires a.Length>=1
requires 0<=low<=high<=a.Length
decreases high - low
modifies a
{
   if (high-low > 1) // need 2 or more items to sort
   {
       var pivi := Partition(a, low, high); // notice the bounds
       QuickSort(a, low, pivi);             // lower part
       QuickSort(a, pivi+1, high);          // upper part
   }
}
```

```
method Partition (a:array<int>, low:int, high:int) returns (pivi:int)
requires a.Length>0
requires 0<=low<high<=a.Length
ensures  0<=low<=pivi<high<=a.Length
modifies a
{ pivi := low;
  var i := low+1;
  while (i < high)
  invariant low<=pivi<i<=high
  { if (a[i] < a[pivi])
        {  var stor := a[i]; // 1st smaller item
           a[i]  := a[i-1];
           var back := i - 1;
           while (back > pivot)
           invariant a[pivi] > stor
           {   a[back+1] := a[back];
               back := back-1
           }
           a[pivi+1] := a[pivi];
           a[pivi] := stor;
           pivi := pivi+1;
        }
    i := i+1;
} }
```



Verifying quicksort

- We need a few predicates
    - Our trusted sorted
    - Our trusted pivot to check left < pivot, right > pivot
    - Sandwich

```
predicate Sorted (a:array<int> ,lo:int , hi:int )
requires 0<=lo<=hi<=a.Length
reads a
{  forall j,k:: lo<=j<k< hi ==> a[j] <= a[k]  }

predicate Sandwich (a:array<int>, lo:int, hi:int) // a[lo-1] <= a[lo..hi-1] < a[hi]
reads a
{ (0<lo<=hi<=a.Length ==> forall j:: lo<=j<hi ==> a[lo-1]<=a[j]) &&
  (0<=lo<=hi<a.Length ==> forall j:: lo<=j<hi ==> a[j]<a[hi]) }

predicate PivotOrder (a:array<int>, lo:int, pivi:int, hi:int)
requires 0<=lo<=pivi<hi<=a.Length
reads a
{ (forall j:: lo<=j<pivi ==> a[j]<a[pivi]) && // everything to the left is smaller
  (forall j:: pivi<j<hi ==> a[pivi]<=a[j]) }  // everything to the right is larger

method QuickSort (a:array<int>, low:int, high:int)
requires a.Length>=1
requires 0<=low<=high<=a.Length
requires Sandwich (a, low, high)
ensures  Sandwich (a, low, high)
ensures  Sorted  (a, low, high)
decreases high-low
modifies a
{
   if (high-low > 1)
   {
       var pivot := partition(a, low, high);
       QuickSort(a, low, pivot);
       QuickSort(a, pivot+1, high);
   }
}

method Partition (a:array<int>, low:int, high:int) returns (pivi:int)
requires a.Length>0
requires 0<=low<high<=a.Length
requires Sandwich (a, low, high)

ensures  Sandwich (a, low, high)
ensures  0<=low<=pivi<high<=a.Length
ensures  PivotOrder (a, low, pivi, high)

modifies a

{  pivi := low;
   var i := low+1;

while (i < high)
   invariant low<=pivi<i<=high
   invariant PivotOrder (a, low, pivi, i)
   invariant Sandwich (a, low, high)
   {  if (a[i] < a[pivi])
      {  var stor := a[i];
         a[i] := a[i-1];
         var back := i-1;
         while (back > pivi)
         invariant a[pivi] > stor
         invariant PivotOrder(a, low, pivi, i+1)
         invariant Sandwich (a, low, high)
         {  a[back+1] := a[back]; back := back-1; }
         a[pivi+1] := a[pivi];
         a[pivi] := stor;
         pivi := pivi+1;
      }
      i := i+1;
}
```

Dynamic framing

- Built upon by Rustan Leino, Herbert and Quaresma during a summer school guided by Leino.
- Verifying dynamic classes that require runtime behavior to be verified can be very difficult.
    - Makes things like inheritance impossible

Very simple unverified counter

Simple and unverified boring counter

```
class Counter // Dafny 2.3.0
{  var value: int;
   constructor()
   { value := 0; }

   method Inc() modifies this;;
   { value := value + 1;}

   method Dec() modifies this;
   { value := value - 1;}

   method GetCounter() returns (x: int)
   { return value;}
}
```

```
method Main()
{
   var c := new Counter();
   c.Inc(); c.Inc(); c.Dec(); c.Inc();
   var i: int := c.GetCounter();
   print "counter = ", i;
}
```

```
Program verifier . . .  7 verified, 0 errors
Program compiled successfully
Running...
counter = 2
```

However, we can add more to make this interesting. We can use 2 counters, one counting the number of increments and another counting the number of decrements and abstract away from the value, by making the value the increments – decrements

```
class Counter // Dafny 2.3.0
{ var incs: int, decs: int;

  constructor Init()
  { incs, decs := 0, 0; }

  method GetCounter() returns (x: int)
  { x := incs - decs; }

  method Inc() modifies this;
  { incs := incs + 1;}

  method Dec() modifies this;
  { decs := decs + 1;}
}
```

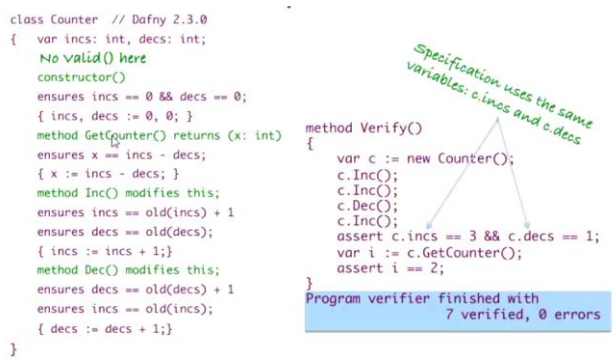
```
method Main()
{
   var c := new Counter();
   c.Inc(); c.Inc(); c.Dec(); c.Inc();
   var i: int := c.GetCounter();
   print "counter = ", i;
}
```

```
Program verifier . . .  7 verified, 0 errors
Program compiled successfully
Running...
counter = 2
```


But notice, there is no verification yet

The class is missing a valid invariant so therefore it is incomplete.

We can use ghost variables to define behavior on the class. We can use a ghost counter to make sure that the incs – decs is the counter



In the counter example, the class invariant is:

```
predicate Valid()  reads this;
{ value := incs - decs }
```

Every method except the constructor, requires valid ensures valid.

```
class Counter  // Dafny 2.3.0
{  ghost var shadow:int;
   var incs:nat, decs:nat;
   predicate Valid() reads this;
   { shadow == incs - decs }
   constructor () // 1.9.7 needs 'modifies this'
   ensures Valid()
   ensures shadow == 0
   { incs, decs, shadow := 0, 0, 0; }
   method Inc() modifies this
   requires Valid(); ensures Valid()
   ensures shadow == old(shadow) + 1
   { shadow := shadow + 1; incs := incs + 1; }
   method Dec() modifies this
   requires Valid(); ensures Valid()
   ensures shadow == old(shadow) - 1
   { shadow := shadow - 1; decs := decs + 1; }
   method GetCounter() returns (r:int)
   requires Valid(); ensures Valid()
   ensures r == shadow
   { r := incs - decs; }
}
```

```
method Verification()
{
  var c := new Counter();
  c.Inc();
  c.Inc();
  c.Dec();
  c.Inc();
  var i := c.GetCounter();
  assert i == c.shadow == 2;
}
```

```
Dafny program verifier finished
        with 8 verified, 0 errors
```

Here we use the initial method of counting as the ghost variable to verify our abstract methods.

The reads and modifies clauses describe the frame, what is visible to the method. These are also dynamic for object verification. If we have an object inside an object then reads and modifies complicates

Here is the new class:

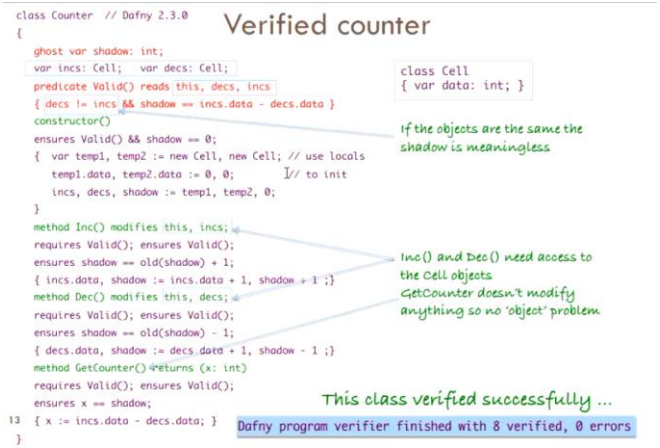
```
class Cell
{ var data: int; }
```

We modify the class Counter as follows:

```
class Counter
{
   ghost var shadow: int;
   var incs: Cell;
   var decs: Cell;
   …
```

This is the first time we have two classes interacting. The counter uses the cell.

- The predicate and methods must now work with incs and decs as objects
- Data is referred to by cell.data
- The objects incs and decs have to be added to reads and modifies clauses in valid and inc() and dec()



notice how we say decs != incs in the Valid(). If that were the case the shadow would be useless.

However, once we try to verify with real values, we will get errors. This is because we didn't specify what we read and modify inside the Cell. We need to include inc and dec in the modifies clause for the cell objects

```
method Verification()
{
   var c := new Counter();
   c.Inc(); c.Inc(); c.Dec(); c.Inc();   // line 43
   var i := c.GetCounter();
   assert i == c.shadow == 2;
}
stdin.dfy(43,8):  Error: call may violate context's modifies clause
stdin.dfy(43,17): Error: call may violate context's modifies clause
stdin.dfy(43,26): Error: call may violate context's modifies clause
stdin.dfy(43,35): Error: call may violate context's modifies clause
Dafny program verifier finished with 7 verified, 4 errors
```

Problem here is with authenticity. We need to know the initial cells are fresh, otherwise we can just assign any cell with any values. Also we need to ensure that the objects don't change so we don't ruin the cell

The problem is '*authenticity*'

- when the constructor is called
  - Dafny does not know where the object that is returned has come from
  - solution, use *fresh()*
- when *Inc()* and *Dec()* are called:
  - the first call might change the data in *incs* or *decs* to anything …
  - … and a second call might then modify some unknown thing

Verified counter with named objects

```
class Cell  // Dafny 2.3.0
 {  var data: int; }

 class Counter
 {
   ghost var shadow: int;
   var incs: Cell;
   var decs: Cell;

   predicate Valid() reads this, incs, decs;
   { decs!=incs && shadow == incs.data - decs.data }

   constructor()
   ensures Valid();
   ensures shadow == 0;
   ensures fresh(decs) && fresh(incs)
   {  var temp1, temp2 := new Cell, new Cell;
      temp1.data, temp2.data := 0, 0;
      incs, decs, shadow := temp1, temp2, 0;
   }

   method Inc() modifies this, incs
   requires Valid(); ensures Valid()
   ensures shadow == old(shadow) + 1
   ensures decs == old(decs) && incs == old(incs)
   { incs.data, shadow := incs.data + 1, shadow + 1 ;}

   method Dec() modifies this, decs
   requires Valid(); ensures Valid()
   ensures shadow == old(shadow) - 1
   ensures decs == old(decs) && incs == old(incs)
   { decs.data,shadow := decs.data+1, shadow-1 ;}

   method GetCounter() returns (x: int)
   requires Valid(); ensures Valid();
   ensures x == shadow;
   { x := incs.data - decs.data; } }
```

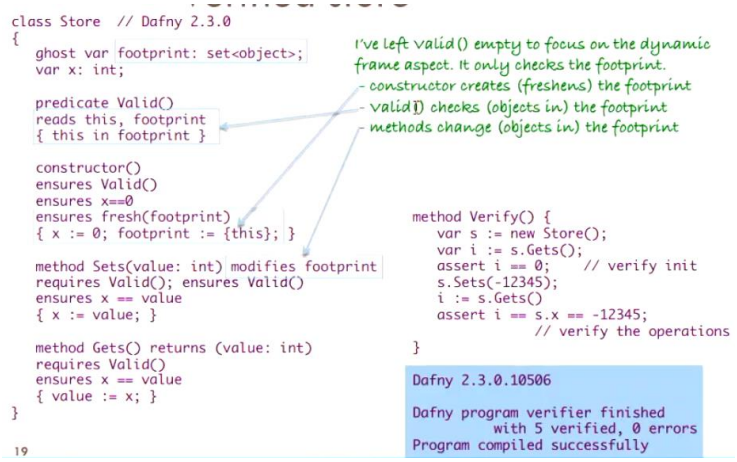
Now we will have told dafny, don't worry the objects won't be corrupted with the method and will verify correctly.

```
method Verification()
{
   var c := new Counter();
   c.Inc(); c.Inc(); c.Dec(); c.Inc();
   var i := c.GetCounter();
   assert i == c.shadow == 2;
}
```

```
Dafny 2.3.0.10506

Dafny program verifier finished with 10 verified, 0 errors
Program compiled successfully
```

This is a bit hacky and adhoc, our modifies and reads clause is very cluttered and can be very messy. There are ways to make this abstract by using a footprint. This will also get rid of the security risk where we don't name what is changing, since it can be easy to corrupt data in another object without dafny knowing about it.

Our footprint is a set<object> (a set of objects). And we read and modify the footprint. The footprint contains all the objects in the class including itself.



Notice how we have reads this and footprint in Valid. But footprint is this, so why do we need reads this? The answer is because that footprint is a MEMBER of the class, so we need to read the class to first see footprint.

When we use footprint inside of our counter class now we get the following

We have 2 state variables, *incs* and *decs*, each of type *Cell*

- they exist as objects

There are 3 objects in the program: *this, incs, decs*

- Constructor must create a *fresh* footprint containing these objects

```
ensures fresh(footprint);
footprint := {this, incs, decs};
```

- *Valid() reads* the footprint as every method 'requires' it

```
reads this, footprint;
{this, incs, decs} <= footprint // this is subset
```

- Methods *Inc() and Del()* will modify objects, so may change the footprint
  - Any new objects created here will be fresh

```
modifies footprint
ensures fresh(footprint - old(footprint));
```

Note <= is subset operator

Here is the final counter class

```
class Cell  // Dafny 2.3.0
{  var data: int; }

class Counter
{
   ghost var shadow: int;
   ghost var footprint: set<object>;

   var incs: Cell;
   var decs: Cell;

   predicate Valid()
   reads this, footprint;
   {
      {this, incs, decs} <= footprint &&
      incs != decs  && shadow == incs.data - decs.data
   }

   constructor()
   ensures Valid();
   ensures shadow == 0;
   ensures fresh(footprint);
   {
      var temp1, temp2 := new Cell, new Cell;
      temp1.data, temp2.data := 0, 0;
      incs,    decs    := temp1, temp2;
      shadow := 0;
      footprint := {this, incs, decs};
   }

   method Inc() modifies footprint;
     requires Valid(); ensures Valid();
     ensures shadow == old(shadow) + 1;
     ensures fresh(footprint - old(footprint));
     { incs.data := incs.data + 1;
       shadow := shadow + 1; }

   method Dec() modifies footprint;
     requires Valid(); ensures Valid();
     ensures shadow == old(shadow) - 1;
     ensures fresh(footprint - old(footprint));
     { decs.data := decs.data + 1;
       shadow := shadow - 1; }

   method GetCounter() returns (x: int)
     requires Valid();
     ensures x == shadow;
     { x := incs.data - decs.data; }
}
```

```
method Verification()
{
   var c := new Counter();
   c.Inc();
   c.Inc();
   c.Dec();
   c.Inc();
   var i: int := c.GetCounter();
   assert i == c.shadow == 2;
}
```

```
Dafny 2.3.0.10506

Dafny program verifier finished with 8 verified, 0 errors
Program compiled successfully
```

Now let's add a cheat method called jump which will add/subtract 'x' from our counter.

24

Here is the new method:

```
method Jump(num: int) modifies footprint;
requires Valid(); ensures Valid();
ensures shadow == old(shadow) + num;
ensures fresh(footprint - old(footprint));
{
   if (num < 0) {decs.data := decs.data + num; }
   if (num > 0) {incs.data := incs.data + num; }
   shadow := shadow + num;
}
```

```
method Verification()
{
  var c := new Counter();
  c.Inc(); c.Inc(); c.Dec(); c.Inc();
  var i:= c.GetCounter();
  assert i == c.shadow == 2;
  c.Jump(8);
  i := c.GetCounter();
  assert i == c.shadow == 10;
  c.Jump(-10);
  i := c.GetCounter();
  assert i == c.shadow == 0;
  c.Jump(0);
  i := c.GetCounter();
  assert i == c.shadow == 0;
}
```

Easy.

Verifying a Cake

The class Cell was trivial: `class Cell{ var data: int; }`

What happens if this class, which we'll call the base class, and the class that uses it, both contain constructors/methods/*Valid()*?

When an object of the base class is created in the using class:

- the object must be added to the footprint (e.g. the constructor)
- changes to the object must be recorded in the footprint (in methods)
- the base class *Valid()* must be maintained (by *Valid()* in the using class)
  - the base class has its own footprint of course.

When we have an object inside an object, the outer object must verify the inner object before it verifies itself. Below is the inner class with its own verification

```
// Base class
class Cake // Dafny 2.3.0
{
   ghost var footprint: set<object>;
   // other fields go here

   predicate Valid() reads this, footprint
   {
      this in footprint
      // something more here
   }

   constructor()
   ensures Valid()
   // postcondition for init code
   ensures fresh(footprint)
   {
      // init code
      footprint := {this};
   }

   method Ice() modifies footprint
   requires Valid(); ensures Valid()
   // postconditions of code
   ensures fresh(footprint - old(footprint))
   {
      // code here
   }
}
```

Now let's make a class that uses the cake class. Note we will need to verify that the cake class is valid in the outer class valid.

```
class Sort { // Dafny 2.3.0
   var mud: Cake;
   // no data defined to keep it simple
   ghost var footprint: set<object>;

   predicate Valid() reads this, footprint
   {  {this, mud}   <= footprint &&    // subset
      mud.footprint <= footprint &&    // mud's footprint is subset
      this !in mud.footprint     &&    // this cannot be in both
      mud.Valid()                      // mud must be valid
      // no class fields to verify
   }
   constructor()
   ensures Valid()
   ensures fresh(footprint)
   {  var temp := new Cake();
      mud := temp;
      footprint := temp.footprint;
      footprint := {this} + footprint; // union
   }
   method Finish() modifies footprint
   requires Valid(); ensures Valid()
   ensures fresh(footprint - old(footprint))
   {  mud.Ice();
      footprint := footprint + mud.footprint;
   }
}
```

```
Dafny 2.3.0.10506

Dafny program verifier finished
        with 8 verified, 0 errors
Program compiled successfully
```

The end <3