Datatypes

For instance to define a binary tree:

```
datatype Tree = Leaf | Node(Tree, Tree)
```

- Declare type

  ```
  datatype D<T> = Ctor1 | Ctor2 | ...

  datatype List<T> = Nil | Cons(head: T, tail: List<T>)
  datatype Semaphore = Green | Yellow | Red
  ```

- Construct instance

  ```
  var list1 := Cons(5, Nil); var sem1 := Green;
  ```

- Update instance

  ```
  d[CtorParam := Value]

  list1 := list1[head := 1];
  ```

- Constructor check

  ```
  d.Ctor?
  ```

- Field selector

  ```
  d.CtorParam

  function Length(x: List<T>) : nat {
  if x.Cons? then 1 + Length(x.tail) else 0
  }
  ```

- Case analysis:

  Match expression (for a match statement, use "=> stmt;" instead of "=> expression") {} are optional

  ```
  function Length(x: List<T>) : nat { match x {
  case Nil => 0
  case Cons(h, t) => 1 + Length(t) } }
  ```

Copy tree values to an array

```
type T = int
datatype Tree<T> = Leaf | Node(Tree, T, Tree)
method Fill(t: Tree<T>, a: array<T>, start: int)
                returns (end: int)
{
match t {
    case Leaf => end := start;
    case Node(left, x, right) =>
      end := Fill(left, a, start);
    if end < a.Length {
        a[end] := x;
        end := Fill(right, a, end + 1);
    }
}
}
```

Verify the correctness of Fill

```
function contains<T(==)>(t: Tree<T>, v: T): bool
{
match t {
case Leaf => false
case Node(left, x, right) =>
x == v || contains(left, v) || contains(right, v)
}
}

}

method Main()
{ var q := Node(Node(Leaf,4,Leaf),5, Node(Leaf,3,Leaf));
  var t := new int[10];
  var n := Fill(q,t,0);
  assert contains(q,3);
  print t[0], t[1], t[2];
}
```

Pre e Pos Condições

```
method Fill(t: Tree<T>, a: array<T>, start: int)
          returns (end: int)
requires 0 <= start <= a.Length
decreases t
```

```
modifies a
ensures  start <= end <= a.Length
ensures forall i:: 0 <= i < start ==> a[i] == old(a[i])
ensures forall i:: start <= i < end ==> contains(t, a[i])
```

Field selectors/Destructors

To access the fields of a datatype one can name them

```
type T = int
datatype Tree<T> = Leaf | Node(left:Tree, key:T, right:Tree)
```

and

```
function countLeaf(t: Tree<T>): nat {
 if t.Leaf? then 1
 else countLeaf(t.left) + countLeaf(t.right)
}
```

Lists

Lemmas and Induction

```
predicate OddPre(n: nat) {
    n%2 == 1
}

function Odd(n:nat): bool
decreases n
{ if n == 1 then true
else if  n == 0 then false
else !Odd(n-1)
}



function OddTail(n:nat,o:bool): bool
decreases n
{ if n == 1 then o
else if  n == 0 then ! o
else OddTail(n-1, !o )
}

lemma {:induction} TestOdd(n: nat)
decreases n
```

```
ensures Odd(n) == OddPre(n)
{}

datatype Nat = Z | S(Nat)

function   Plus(x:Nat, y:Nat): Nat
decreases x;
{
match x {
    case Z => y
    case S(z) => S(Plus(z,y))
}
}

lemma {:induction n} example(n:Nat)
decreases n;
ensures Plus(n,Z) == n
{}

lemma {:induction false} example0 (n: Nat)
decreases n
ensures Plus(n,Z) == n
{
    match n {
    case Z => { assert Plus(Z,Z) == Z;}
    case S(k) => {
        assert Plus(n,Z) == S(Plus(k,Z));
        example0(k);
        assert S(Plus(k,Z)) == S(k);
        assert  S(k) == n;
        }
    }

}
```

DbC: design by contract

- Concepts introduced by Bertrand Meyer for the Eiffel language in 1980's
- A *contract* of a procedure or object is a pair (precondition, postcondition) such that

  precondition : which proof obligations required by the invoker expressing input conditions or or the initial state of an object

  postconditions beneficts expected when the procedure terminates or the final state of an object

- a contract is violated if who invokes does not respect the preconditions or if the postconditions are not satisfied

```
class interface ACCOUNT

create
    make

feature

    balance: INTEGER
        ...

    deposit (sum: INTEGER)
            -- Deposit sum into the account.
        require
            sum >= 0
        ensure
            balance = old balance + sum

    withdraw (sum: INTEGER)
            -- Withdraw sum from the account.
        require
            sum >= 0
            sum <= balance - minimum_balance
        ensure
            balance = old balance - sum

    may_withdraw ...

end -- ACCOUNT
```

Objects in Dafny

- an object is an instance of a class
- it is accessed by reference and is stateful
- the class defines state components of an object, *fields*, and operations, *methods*: *members*of the class.

Class invariants (Objects)

- A *class invariant* is a condition that defines the valid states (members values) of the objects (class instances).
- Besides pre/post conditions each method, inside a class, has to respect the class invariant:
  - Each constructor needs to ensure that all invariants hold at the end of its execution.

– Each method needs to ensure that all invariants hold at the end of its execution, assuming that they hold in the beginning.

```
class
```

```
class Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    constructor (cap: nat)
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate  isEmpty()
    {
        size == 0
    }
    predicate   isFull()
    {
        size == elems.Length
    }
    method push(x : T)
    {
        elems[size] := x;
        size := size + 1;
    }
    function   top() : T
    {
         elems[size-1]
    }
    method pop()
    {
         size := size-1;
    }
}
```

Objects

- fields: `elems, size`
- constructor: called when objects are created (initialization)
- methods: `method`

- initialization: `var s:= new Stack(3)`

```
Stack

method {:verify false} testStack()
{
    var s := new Stack(3);
    assert s.isEmpty();
    s.push(1);
    s.push(2);
    s.push(3);
    assert s.top() == 3;
    assert s.isFull();
    s.pop();
    assert s.top() == 2;
}
```

Objects

- classes (or *trait*) can extend other classes
- Methods can be declared static
- An object is referred by `this`
- and their members by ".": `this.size` (but `this` can be omitted)
- A variable of a class (instance) can be `null` if declared with `?` after the class name
- construct can have names
- The type of all classes is `object`
- Immutable fields can be declared with `const`
- Classes may include `predicates` or `functions`.

```
class A {
 constructor()
}

method main(){
var a: A := new A();
var o:object = a;
var n:A? = null;
}
```

Class invariant- `Valid`

- predicate `Valid()` describes the class invariant
- this invariant is required (precondition) for all methods/functions (except the constructors)
- and needs to hold at the end of the execution (postcondition) by all constructors and modifiers (any method)
- with the attribute `:autocontracts` Dafny adds authomatically `requires/ensures` for `Valid()` and other required contracts.

***framing* for objects**

- Object are mutable and stored in the *heap*
- use *dynamic frames*
- It is necessary to state for *function* (or *predicate*) which parts are read `reads` and for *method* which parts are modified `modifies` ()
- if only one field of an object is read that van be expressed with ‘ : `reads this‘Value`

Counter

```
class Counter {
var Value: int;

predicate Valid()
reads this‘Value
{
Value >= 0
}
constructor Init ()
ensures Value == 0
ensures Valid()
{ Value := 0;}
method GetValue() returns (x: int)
modifies this
requires Valid()
ensures x == Value
{x := Value;}
```

Counter

```
method Inc ()
 modifies this
 requires Valid()
 ensures Value == old(Value)+1
 ensures Valid()
 {Value := Value + 1;}

method Dec ( )
modifies this
requires Value > 0 && Valid()
ensures Value == old ( Value ) -1 && Valid()
{Value := Value -1;}
}
```

Allocation of new objects

- if during the execution of a method new objects are allocated in the *heap* that should be indicated using `fresh`.

```
method NewSorted(n: nat) returns (v: array<int >)
ensures fresh ( v )
ensures Sorted ( v )
ensures v.Length = n
{}
```

Invariant of `Stack`

```
ghost predicate Valid()
reads this
{
size <= capacity == elems.Length
}

constructor (cap: nat)
    requires cap > 0
    ensures fresh(elems)
    ensures Valid()
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
```

```
predicate isEmpty()
    reads this
    {
        size == 0
    }

predicate isFull()
    reads this
    {
        size == elems.Length
    }

method  push(x : T)
    modifies this, elems
    requires Valid() && size < capacity
    {
        elems[size] := x;
        size := size + 1;
    }

function  top(): T
    requires Valid() && size > 0
    reads this, elems
    {
         elems[size-1]
    }

method pop()
    requires Valid() && size > 0
    modifies this
    ensures Valid()
    ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}


:autocontracts

class{:autocontracts} Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    ghost predicate Valid()
```

```
    reads this
    { size <= capacity == elems.Length
    }
    constructor (cap: nat)
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate method  isEmpty()
    {
        size == 0
    }
    predicate method isFull()
    {
        size == elems.Length
    }


:autocontracts


    method push(x : T)
    requires size < capacity
    modifies this, elems
    ensures elems[..size]==old(elems[..size])+[x]
    {
        elems[size] := x;
        size := size + 1;
    }

    function  top(): T
    requires size > 0
    {
         elems[size-1]
    }
    method  pop()
    requires size > 0
    modifies this
    ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}
```

Inheritance

A subclass cannot break the contract of their superclass, but

- precondition can be weakned
- Postcondition can be strengthened

Example: The class `Circle` inherits from the `Shape` class.

```
trait  Shape
{
    var center: (real, real);

    ghost predicate Valid()
      reads this

    function  getSizeX(): real
       requires Valid()
       reads this

    function  getSizeY(): real
       requires Valid()
       reads this

    method resize(factor: real)
      requires factor > 0.0 && Valid()
      modifies this
      ensures Valid()
      ensures getSizeX() == factor * old(getSizeX())
      ensures getSizeY() == factor * old(getSizeY())
      ensures center == old(center)
}

class Circle extends Shape
{
    var radius: real;
    predicate Valid()
      reads this
    {
        radius > 0.0
    }
    constructor Circle(center: (real, real), radius: real)
      requires radius > 0.0
      ensures this.center == center && this.radius == radius && Valid()
    {
        this.center := center;
```

```
        this.radius := radius;
    }
    function  getSizeX(): real
      requires Valid()
      reads this
    {
        radius
    }
    function  getSizeY(): real
      requires Valid()
      reads this
    {
        radius
    }
}
```

Subclasse

```
    method resize(factor: real)
      requires factor != 0.0 && Valid()
      modifies this
      ensures center == old(center)
      ensures radius == abs(factor) * old(radius)
      ensures Valid()
    {
       radius := abs(factor) * radius;
    }

    function abs(x: real): real
    {
        if x >= 0.0 then x else -x
    }
}
```

See that

$$factor > 0.0 \rightarrow factor \neq 0.0$$
$$radius == abs(factor) * old(radius) \rightarrow getSizeX() == factor * old(getSizeX())$$