7.- Arrays and Framing

- Arrays are objects on the heap, hence mutable.
  Arrays are sequences of mutable locations.
- Arrays are dynamically allocated objects, hence they can be null.
  - `array<T>` is the type of non-null arrays of elements of type `T`.
  - `array?<T>` is the type of possibly null arrays of elements of type `T`.
- Any `a: array<T>` have a built-in length field: `a.Length` that is inmutable.

- Element access uses the standard bracket syntax: `a[0]`, `a[1]`, `a[2]`,... and indexes start by `0` and ends by `a.Length-1`.
- The element stored at `i` can be changed to a value `t` using the array update statement: `a[i] :=t;`
- All array accesses must be proven to be within bounds, therefore invariants on bounds for array-indexes are often required.
- Because bounds checks are proven at verification time, no runtime checks need to be made.

- One-dimensional arrays support operations that convert a stretch of consecutive elements into a sequence:
  For any `a: array<T>`, any pair of integers `lo` and `hi` satisfying `0 ≤lo ≤hi ≤a.Length`, the following operations each yields a `seq<T>`:

expression	description
`a[lo..hi]`	subarray conversion to sequence
`a[lo..]`	drop
`a[..hi]`	take
`a[..]`	array conversion to sequence

Creating new arrays

- Non-ghost methods are allowed to allocate new objects and modify their state.
- To create a new object (e.g. an array), it must be allocated with the `new` keyword.
  - `var a :=new T[n];` create an `array<T>` a of length `n`
  - requires `n` to be non-negative integer
- `new T[n](f)` check that
  - `f : int` $\rightarrow$ `T` and
  - for all `i` such that `0` $\leq$ `i` $<$ `n`: `i` satisfies the requires of `f`.
- In `new T[n](f)` the function `f` can be define locally by an expression `i => E`, for example `i => i*2`.
- Type-parameter suffix in Dafny: `(0)`.
  - `T(0)` restrict type `T` to allow non-initializable values.
  - To use more than one of them either $\texttt{T}^{(=)}\texttt{(0)}$ or `T(=,0)`.

Multidimensional arrays

- Arrays can also be multidimensional: `array2<T>`, `array3<T>`, ...
- For multidimensional arrays, notation is similar, e.g.
  `matrix :=new T[m, n];` creates `matrix: array2<T>`.
- Lengths can be retrieved using the immutable fields `Length0` and `Length1`.
  For example, the following holds of the array created above:
  `matrix.Length0 =m ∧matrix.Length1 =n`.
- Higher-dimensional arrays have inmutable lengths fields:
  `Length0, Length1, Length2, . . .`
- No operation to convert stretches of elements from a multi-dimensional array to a sequence.

Framing

- Arrays are objects with an state which is mutable.
- A *frame* is specified by a *set of object* references.

```
modifies S + {r};
modifies S, {r};
modifies S, r;
modifies S; modifies r;
```

```
reads S + {r};
reads S, {r};
reads S, r;
reads S; reads r;
```

- The `modifies` and `reads` clauses govern modifications (in a method) and dependencies (of a function), respectively. Each specifies a frame.

Methods/Functions Framing

- Methods are allowed to read whatever they want, so these reads do not need to be specified.
- Functions/Predicates are not allowed to modify objects, so modifies can not be specified.
- The `modifies` clause says that the method has license to modify the state of any of those objects.
- The `reads` clause says that the function is allowed to depend on the state of any of those objects.

The `old` keyword/function

- Mutable objects: the postcondition must be expressed in terms of the state of the variables before and after method execution.
- The state/value of mutable objects is loaded in the heap.
- The `old` keyword, when applied to a variable (`old(variable)`) operates as a function which refers to the value of the `variable` at the time the method was invoked.
- `old` only affects (makes sense on) values looked up in the heap.
- Example: For a sorting algorithms, with an in-parameter `a: array<int>`, the sequence `a[..]` of elements of the array `a` (which are stored in the heap) is a permutation of `old(a[..])`, although `old(a) =a`.

The `fresh` keyword/predicate

It is sometimes important for the verifier to know that some given object has been freshly allocated in a given method.

```
method createArray () returns (a:array<int>)
    ensures fresh(a)
{
a := new int[6];
}
```