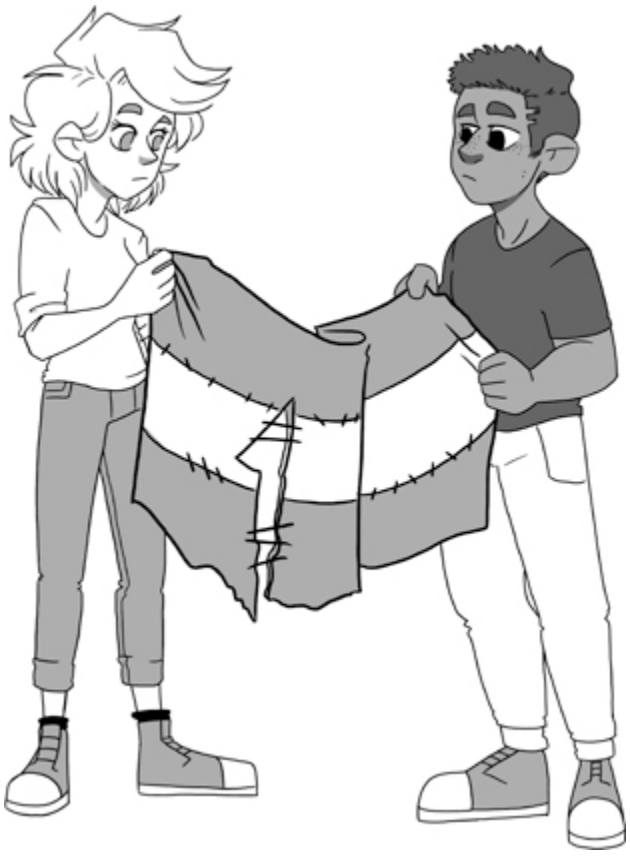
Chapter 15

In-situ Sorting



In Chapter 8, I covered specifications and programs for sorting inductive lists. Since lists are immutable, those programs necessarily returned sorted versions of their inputs, without modifying the input. In this chapter, we'll consider programs that reorder the elements of a given array to make it sorted. That is, the sorting is done *in situ*, Latin for “in place”.

15.0.Dutch National Flag

I'll start us with a problem of sorting an array with three kinds of values. The values are the three colors red, white, and blue, and the order in which we'll sort them is what has given this problem its name, the Dutch National Flag (first red, then white, then blue).

We'll start by introducing the colors and a way to compare them.

```
datatype Color = Red | White | Blue



ghost predicate Below(c: Color, d: Color) {
```

```
c == Red || c == d || d == Blue

}
```

Since we are going to sort the array in place, the only parameter to our method is a reference to the array. The write frame says the method can modify the contents of the array. As we saw in Section 8.0, the postcondition has two parts. One part says the elements are sorted on exit, and the other part says that the multiset of elements in the array is the same before and after the method. Using the notation we saw in Section 13.4, we can write the latter using a conversion from the array elements to a sequence to a multiset:

```
method DutchFlag(a: array<Color>)

modifies a

ensures forall i, j :: 0 <= i < j < a.Length ==> Below(a[i], a[j])

ensures multiset(a[..]) == old(multiset(a[..]))
```

The idea behind the algorithm is to structure the given array into four adjacent segments. The first segment will contain only red elements, the next segment only white elements, and the last segment only blue elements. The remaining segment (which I'll place between the white and blue segments, though another choice would be to place it between the red and white segments) will contain the elements we haven't yet sorted into its proper segment.

To keep track of the four segments, we'll need three markers. These markers will play the role of our loop indices, and I will name them `r`, `w`, and `b`. The following diagram summarizes the ideas so far. In fact, the diagram very much expresses the invariant of the loop we'll write.

Here's the loop specification:

```
{

var r, w, b := 0, 0, a.Length;

while w < b

invariant 0 <= r <= w <= b <= a.Length

invariant forall i :: 0 <= i < r ==> a[i] == Red

invariant forall i :: r <= i < w ==> a[i] == White

invariant forall i :: b <= i < a.Length ==> a[i] == Blue

invariant multiset(a[..]) == old(multiset(a[..]))

}
```

The initial assignments of `r,w,b` make the three colored segments empty, leaving all array elements in the unsorted segment. The loop guard says to continue iterating as long as there are elements in that unsorted segment. When the unsorted segment is empty, the invariant and the negation of the guard imply `w == b`, which says the array consists solely of the three colored segments in the correct order. The final invariant uses the *always* Loop Design Technique 13.1.

By the way, note the consistent use of half-open intervals in the ranges of the quantifiers. This is a consequence of thinking of `r,w,b` as delineating the segments, and it is also the way we usually represent ranges in computer science. The ranges compose nicely, because two segments are consecutive when the upper limit of one segment is the lower bound of the next (see also Sidebar 13.0). Indeed, when `w == b`, you can directly see that the three quantifiers account for all the array elements.

One task left: implement the loop body. Each iteration will sort one more element, so we need to inspect one of the elements in the unsorted segment. The obvious candidates are `a[w]` and `a[b-1]`, since those elements are adjacent to a sorted region. We'll use `a[w]`, which gives us the following structure of the loop body:

```
{
match a[w]
case Red => ?
case White => ?
case Blue => ?
}
```

To fill in the three question marks, you may find the diagram representation of the invariant from above useful.

The simplest case is when `a[w]` is white. Then, all we need to do is include it in the white region, which we do by incrementing `w`:

```
case White =>
w := w + 1;
```

If `a[w]` is red, we want to move it into the red region. But the white region starts where the red region ends, so we'll need to move the first white element to the end of the white region:

```
case Red =>
a[r], a[w] := a[w], a[r];
r, w := r + 1, w + 1;
```

In the event that the white region is empty—that is, if `r == w`—the swap of `a[r]` and `a[w]` will amount not moving any elements, but that is still fine.

Finally, if `a[w]` is blue, we move it to `a[b-1]` and decrement `b`. This will put the blue element into the blue region and make progress toward termination. However, if we only did this much, then we would break the last invariant, because we would duplicate `a[w]` and clobber the element previously stored in `a[b-1]`. So, it is important to swap `a[w]` and `a[b-1]`:

```
case Blue =>

a[w], a[b-1] := a[b-1], a[w];

b := b - 1;
```

This completes the program.

> **Exercise 15.0.**
>
> Change the invariant to place the unsorted segment between the red and white segments. Then, re-implement the initialization and loop body accordingly.

> **Exercise 15.1.**
>
> Specify a method that sorts an array of booleans. Consider **`false`** to be ordered before **`true`**. Implement the method using a loop that swaps array elements (akin to what we did for the Dutch National Flag).

> **Exercise 15.2.**
>
> Like many representations of sorting algorithms, the `DutchFlag` method above is a simplification of what you would actually do. I represented each element as a color. More typically, the array elements are more complex pieces of data and the color is the just the key that we're sorting the array by. Define a predicate `CBelow` that compares two values based on their color:
>
> ```
> ghost predicate CBelow<T>(color: T -> Color, x: T, y: T)
> ```
>
> and implement a method that sorts an array based the color of its elements.
>
> ```
> method DutchFlagKey<T>(a: array<T>, color: T -> Color)
> modifies a
> ensures forall i, j :: 0 <= i < j < a.Length ==>
> CBelow(color, a[i], a[j])
> ensures multiset(a[..]) == old(multiset(a[..]))
> ```

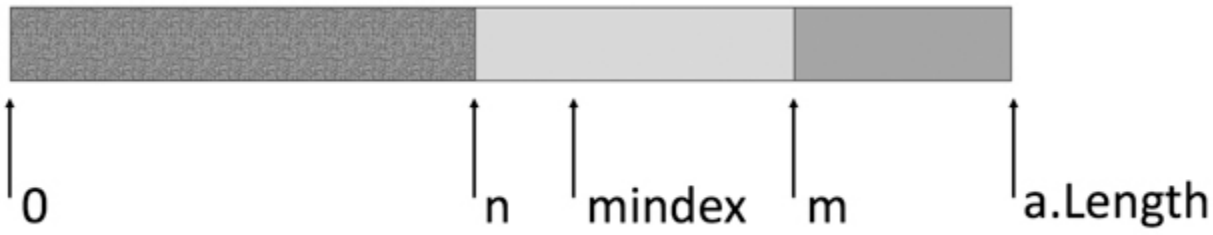
> **Exercise 15.3.**
>
> (Advanced.) Sorting algorithms typically use swapping to rearrange the order of the given elements. But sorting an array of three colors (without any other payloads) can be done without swapping: First, scan the elements of the given array and count the number of occurrences of each color. Second, assign the output according to those three counts, writing the appropriate number of reds, whites, and blues, respectively. Implement and verify the Dutch National Flag method in that way.

15.1.Selection Sort

Selection Sort is a sorting algorithm that repeatedly finds the minimum among the unsorted elements and then appends it to the segment of sorted elements. Its specification is like that of `DutchFlag` in the previous section, but here I'm using integers instead of colors:

```
method SelectionSort(a: array<int>)
modifies a
ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
ensures multiset(a[..]) == old(multiset(a[..]))
```

We'll have two nested loops. With `n` as its loop index, an invariant of the outer loop is that the first `n` elements of the array are sorted. The following diagram illustrates:



The inner loop scans the remainder of the array to find the minimum. As we've seen before, we can specify the minimum as being below all the elements examined and as being one of the elements to be examined. We'll need to keep track of where in the array the minimum is, so we'll use a variable `mindex` for this purpose. By making sure `mindex` is an index into the unsorted part of the array, it follows that `a[mindex]` is one of the elements to be examined. The diagram above sketches this situation, where `m` is the loop index of the inner loop.

That's the main idea. Let's implement the method.

Our starting point is straightforward. We'll use a loop index `n`, we'll turn the first postcondition into an invariant by replacing the constant `a.Length` with variable `n`, and we'll turn the other postcondition into an “always” invariant:

```
{
var n := 0;
while n != a.Length
invariant 0 <= n <= a.Length
invariant forall i, j :: 0 <= i < j < n ==> a[i] <= a[j]
invariant multiset(a[..]) == old(multiset(a[..]))
}
```

Next, we'll work on the inner loop. We've written many loops like it before (including the minimum program in Section 13.3), so we may immediately come up with:

```
var mindex, m := n, n;
while m != a.Length
invariant n <= m <= a.Length && n <= mindex < a.Length
invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
```

```
{
if a[m] < a[mindex] {
mindex := m;
}
m := m + 1;
}
```

We can do a little better. If we make sure `mindex` is less than `m`, then we can write the first invariant as

```
invariant n <= mindex < m <= a.Length
```

This gives us an error, complaining that `mindex < m` does not hold on entry to the loop. But we can fix that by starting `m` off at `n + 1`, which is known not to exceed `a.Length`.

```
var mindex, m := n, n + 1;
```

Indeed, there is no reason to iterate the inner loop for `m == n`, since we initialize `mindex` to `n`, which clearly is the index of the minimum element in `a[n..n+1]`.

Great. After the inner loop, `mindex` tells us where the next smallest element is. So, all we need to do is swap `a[mindex]` into place:

```
a[n], a[mindex] := a[mindex], a[n];
n := n + 1;
```

Something is wrong! The verifier complains that the universally quantified loop invariant is not maintained by the loop. On entry to the loop body, that invariant told us the first `n` elements are sorted and now we are saying the first `n + 1` elements are sorted. What's wrong with that?

One possible culprit is the inner loop. We've seen before (when initializing a matrix in Section 14.1.1) that an inner loop sometimes needs to include some loop invariants of the outer loop. But not here, because our inner loop does not modify the array or anything in the heap; the verifier detects this syntactically, and therefore it is automatically known that the sorted segment of the array remains sorted.

Exercise 15.4.

An alternative way to write the inner loop is to swap any new minimum into `a[n]` as soon as it's detected. This means the inner loop will modify the array, so then the inner loop does need some invariants from the outer loop. After we fix the situation with the current algorithm, change the inner loop as I've just described and verify it. (Arguably, this alternative program is worse, but as an exercise, it will let you get more good practice with invariants.)

Alright, then what? We can check if the verifier still thinks the first `n` elements are sorted, by adding the following assertion immediately after the swap and *before* the increment of `n`:

```
assert forall i, j :: 0 <= i < j < n ==> a[i] <= a[j];
```

The verifier confirms this assertion. This can only mean one thing: the verifier is not convinced that our new `a[n]` is ordered after `a[..n]`.

When you work with program proofs, situations like this arise all the time. We try to write down our design decisions as invariants and these guide our program design. Inevitably, some part of the design remains in our head. Either we thought about the situation correctly, but didn't think to write down the conditions as invariants, or we missed some part of the design that may or may not be correct. Whichever the case may be, the way to make a convincing correctness argument is to formulate what is true as an invariant.

What's missing in our case is the property that the elements in the sorted segment of the array are not just sorted among themselves, but they are also in their final position. Stated differently, the sorted elements are all below the unsorted elements. This is a crucial property of Selection Sort. Let's write it down as an invariant of the outer loop:

```
invariant forall i, j :: 0 <= i < n <= j < a.Length ==> a[i] <= a[j]
```

This property comes in handy in other sorting algorithms, too, so we might as well define a predicate for it:

```
ghost predicate SplitPoint(a: array<int>, n: int)

requires 0 <= n <= a.Length

reads a

{

forall i, j :: 0 <= i < n <= j < a.Length ==> a[i] <= a[j]

}
```

If you choose to introduce this predicate, you can formulate the last invariant as:

```
invariant SplitPoint(a, n)
```

With this invariant, stated directly as a quantifier or via the `SplitPoint` predicate, we have verified Selection Sort.

15.2.Quicksort

The final sorting routine in this book is Quicksort. It performs well in practice, but is noticeably more complicated to verify than the other sorting routines we have considered. The diculty comes down to specifying what parts of the array are modified and what relationships continue to hold in the face of those modifications.

15.2.0.Method specification

Things do start easy, for the specification of `Quicksort` is the same as that of `SelectionSort`:

```
method Quicksort(a: array<int>)

modifies a

ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]

ensures multiset(a[..]) == old(multiset(a[..]))
```

Quicksort divides up its input into smaller segments and sorts those segments recursively. To specify bounds of the segment to be sorted, we'll use an auxiliary method, and the main `Quicksort` method calls the auxiliary method with `0` and `a.Length` as the bounds:

```
{
QuicksortAux(a, 0, a.Length);
}
```

We'll have to work at the specification of `QuicksortAux` for a while. Here are parts of the specification that it's clear we'll need:

```
method QuicksortAux(a: array<int>, lo: int, hi: int)

requires 0 <= lo <= hi <= a.Length

modifies a

ensures forall i, j :: lo <= i < j < hi ==> a[i] <= a[j]

decreases hi - lo
```

We will also need the postcondition that says the multiset of array elements is unchanged, but we'll also need more precise information about which array elements are changed at all. To express `a[i] == old(a[i])` for the indices `i` outside the range from `lo` to `hi`, we'll use a quantifier. We'll have to write this quantifier, and also the multiset property, several times, so let's define a predicate that we can reuse where needed.

15.2.1.Two-state predicates

This predicate is different than any other we've seen so far, because we want it to relate the current state with the pre-state of a method. For this purpose, Dafny provides *two-state predicates*. These are able to read two heaps, the current heap and some heap from the past. The prior heap is accessed with **`old`** expressions, just as in method bodies and postconditions. Calls to a two-state predicate in effect pass in the prior heap. That's all we need to know about two-state predicates to use them in Quicksort.

Here is our two-state predicate:

```
twostate predicate SwapFrame(a: array<int>, lo: int, hi: int)

requires 0 <= lo <= hi <= a.Length

reads a

{

(forall i :: 0 <= i < lo || hi <= i < a.Length ==>

a[i] == old(a[i])) &&

multiset(a[..]) == old(multiset(a[..]))

}
```

Our first use of this predicate is as a postcondition in the specification of `QuicksortAux`:

```
ensures SwapFrame(a, lo, hi)
```

15.2.2.The core algorithm

The idea of Quicksort is simple. Pick an element—known as the *pivot*—among those to be sorted. In terms of the Dutch National Flag, think of the pivot as white, the elements smaller than the pivot as red, and those larger than the pivot as blue. Then, sort according to color, like in the Dutch National Flag algorithm. In Quicksort, this subroutine is usually named `Partition`. Method `Partition` returns the index to the pivot. Then, Quicksort recursively sorts the elements to the left of the pivot and the elements to the right of the pivot.

Here is the body of `QuicksortAux`, which performs the steps I just described:

```
{
if 2 <= hi - lo {
var p := Partition(a, lo, hi);
QuicksortAux(a, lo, p);
QuicksortAux(a, p + 1, hi);
}
}
```

Alas, even after specifying `Partition` (which we'll do below), `QuickSortAux` does not verify as given. The problem is that our `SwapFrame` predicate does not directly give us the information we need about the relation of the elements. We could try to prove some lemmas about `SwapFrame`. Such lemmas would have to be *two-state lemmas*, which I will not cover in this book. Instead, we can use the `SplitPoint` predicate I introduced for Selection Sort at the end of Section 15.1.

15.2.3.Split points

As a final enhancement of the specification of `QuicksortAux`, we will require and ensure that `lo` and `hi` are split points. As a precondition, this says that elements `a[..lo]` are below `a[lo..hi]` and those, in turn, are below `a[hi..]`. As a postcondition, this comes down to saying that the method does not swap any elements across split-point boundaries.

Here's what we add to the specification of `QuicksortAux`:

```
requires SplitPoint(a, lo) && SplitPoint(a, hi)
ensures SplitPoint(a, lo) && SplitPoint(a, hi)
```

15.2.4.`Partition`

It's time to specify and implement `Partition`. In spirit, it is essentially the Dutch National Flag algorithm, but the formulation is different in several ways from what we did in Section 15.0. In the specification, one difference is that we are now sorting only a segment of an array, another is that we don't actually have colors (but see Exercise 15.2), and a third is that `Partition` returns the index of the pivot. In the implementation, one difference is that we need to select the pivot, and another is that it will be easier to keep the pivot separate from the other elements while sorting the “red” and “blue” elements into place.

The specification is big:

```
method Partition(a: array<int>, lo: int, hi: int) returns (p: int)
requires 0 <= lo < hi <= a.Length
requires SplitPoint(a, lo) && SplitPoint(a, hi)
modifies a
ensures lo <= p < hi
ensures forall i :: lo <= i < p ==> a[i] < a[p]
ensures forall i :: p <= i < hi ==> a[p] <= a[i]
ensures SplitPoint(a, lo) && SplitPoint(a, hi)
ensures SwapFrame(a, lo, hi)
```

In addition to the `SplitPoint` and `SwapFrame` predicates like in `QuickSortAux`, this specification says that `p` returns as an index into `a[lo..hi]`. This is important in order for `QuicksortAux` to make progress with each of its recursive calls. The postcondition also says how the elements in the segment `a[lo..hi]` relate to the pivot element `a[p]`.

15.2.5.Implementing `Partition`

We have only one thing left to do for Quicksort, namely to implement method `Partition`. We'll do it in three steps.

As the first step, we select the pivot from among the elements to be partitioned. This selection affects performance and there are heuristics for selecting it well. For the sake of correctness, it does not matter which element we pick, so let's just pick `a[lo]`:

```
{
var pivot := a[lo];
```

As the second step, we use a loop to order the smaller-than-`pivot` elements before the `pivot`-or-larger elements in `a[lo+1..hi]`. Along the lines of the Dutch National Flag, we use two markers, `m` and `n`, so that `a[lo+1..m]` are smaller than `pivot`, `a[m..n]` have not yet been examined, and `a[n..hi]` are `pivot` or larger. Just like the enclosing method specification is big, so is the loop specification:

```
var m, n := lo + 1, hi;
while m < n
invariant lo + 1 <= m <= n <= hi
invariant a[lo] == pivot
invariant forall i :: lo + 1 <= i < m ==> a[i] < pivot
invariant forall i :: n <= i < hi ==> pivot <= a[i]
invariant SplitPoint(a, lo) && SplitPoint(a, hi)
invariant SwapFrame(a, lo, hi)
```

The implementation of the loop examines the next element and either extends the lower segment or

swaps it into the upper segment:

```
{
if a[m] < pivot {
m := m + 1;
} else {
a[m], a[n-1] := a[n-1], a[m];
n := n - 1;
}
}
```

As the third step, we need to insert the pivot into the elements we partitioned. We swap it with the element that's adjacent to the upper region, and then we return the resulting index:

```
a[lo], a[m - 1] := a[m - 1], a[lo];
return m - 1;
}
```

I went through this algorithm with its big specifications quickly. Now that we've reached the end, I suggest that, as a way to continue to familiarize yourself with what just passed in front of your eyes, you go back and temporarily comment out parts of the specifications to see where verification fails.

> **Exercise 15.5.**
>
> One heuristic for selecting a pivot is to look at the leftmost, middle, and rightmost values of `a[lo..hi]` and pick the median of these three values. Add code to do that in `Partition`.

> **Exercise 15.6.**
>
> Write a method that sorts an integer array by inserting all elements into the priority queue we developed in Chapter 10 and then successively removing the smallest remaining element.

15.3.Summary

In this chapter, I showed three algorithms that sort a given array by rearranging its elements. Along the way, we encountered read and write frames, and even two-state predicates. Throughout, we expressed properties of interest using universal quantifiers.

The main point of this and the previous few chapters has been to show you how to reason about algorithms using invariants. When you design a loop, it is crucial that you pay attention to what can be said about the starting and ending state of each iteration. The loop invariant is where you record this design. The loop invariant lets you narrow your focus from the infinitely many iteration behaviors of a loop into the behaviors of a single, but arbitrary, iteration. If you make it clear what you expect to hold at the top of every iteration, you'll have a strong guide to the code you need to write in the loop body.

Invariants are not unique to loops. For example, in Chapter 10, we saw invariants for data structures, and in the chapters to come, we will see invariants for objects.

Here are some exercises for writing your own proofs of sorting-related algorithms. Each of these involves many details, so expect to spend a good amount of time debugging your programs and proofs.

Exercise 15.7.

The central operation in Merge Sort (which we saw in a functional setting in Section 8.2) is the merge operation. Using arrays, specify and implement a method

```
method Merge(a: array<int>, b: array<int>) returns (c: array<int>)
```

that takes two sorted arrays, `a` and `b`, and returns a new, sorted array `c` whose elements are those from `a` and `b`.

Exercise 15.8.

In Section 8.1, we implemented Insertion Sort for a list in the functional setting. For this exercise, specify and implement a method that performs Insertion Sort on a given integer array. Insertion Sort inserts each element, in turn, into a sorted prefix of the array.

You'll need two nested loops. For this exercise, do a swap in each iteration of the inner loop (which means you can use the *always* Loop Design Technique 13.1 for the “same elements” condition).

Exercise 15.9.

Implement Insertion Sort for a given integer array, but (unlike in Exercise 15.8) don't do a swap in the inner loop. Instead, just before the inner loop, save the value of the element to be inserted:

```
var x := a[i];
```

Then, in each iteration of the inner loop, just copy the next element into place:

```
while // …

a[j] := a[j - 1];
```

After the inner loop, write the saved value into place:

```
a[j] := x;
```

These steps will have the effect of rotating the element into place, and it saves time because you only do “half of a swap” with each iteration.

Hint: To prove the “same elements” property, your inner loop will need an invariant like

```
invariant multiset(a[..][j := x]) == multiset(old(a[..]))
```

where the sequence-update expression `s[j := x]` denotes the sequence that is like `s` except with element `j` replaced by the value `x`.

Notes

Quicksort was invented by C. A. R. Hoare, who also introduced the triple notation we've been using in program reasoning. Years after its invention, it was one of the first sorting algorithms to be formally verified [51]. Now, more than half a century later, it is interesting to read that paper's concluding remarks on the prospect of mechanizing such a proof.

An even earlier formal proof was developed for Treesort 3 [88], a version of Heapsort due to Robert W. Floyd.

There are many formal proofs of sorting algorithms. For example, more modern proofs of Insertion Sort, Quicksort, and Heapsort have been formalized in the Coq system [49], and Counting Sort (see also Exercise 15.3) and Radix Sort have been verified in the KeY system [37].

In the Notes section of Chapter 8, I remarked about some functional sorting algorithms. The Why3 gallery of verified programs I mentioned there also contains several imperative sorting algorithms [20] .

An attempt to formally verify TimSort in Java's standard library revealed a bug in the implementation [38]. Informed by the failed proof, the bug was fixed and the corrected algorithm was verified in the KeY system.

In the development of Quicksort in Section 15.2, it was convenient to declare and use the `twostate` predicate `SwapFrame`. While some verifiers support such predicates (e.g., Frama-C [14] supports user-defined predicates that relate two or more memory states), not all do. Fortunately, our `Quicksort` program can easily be written without a name for this predicate, as the next exercise asks you to do:

> **Exercise 15.10.**
>
> In the Quicksort program in Section 15.2, replace all calls to the two-state predicate `SwapFrame` with the body of `SwapFrame` (suitably parameterized), and then remove the declaration of `SwapFrame` altogether. Make sure your new program still verifies.

Chapter 16

Objects

An *object* is a stateful abstraction with identity. In class-based languages (which includes Dafny), each object is created as an instance of a *class*. Access to an object always goes via a *reference* (which is also true of arrays). The class defines state components of its objects, which are called *fields*, as well as operations on the objects, which are functions and methods. Collectively, fields, functions, and methods of a class are called *members* of the class. The identity of an object is determined by its reference, not by the value of its fields.

In this chapter, I'll give three examples that use objects. The examples gradually introduce how we write specifications for and reason about objects. The two essential ingredients are *invariants* of the objects structures and *frames*.

16.0.Checksums

Let's write a class `ChecksumMachine`, whose objects compute checksums for some data. Clients can provide the data in pieces, rather than having to supply all the data to be checksummed at once. For this reason, we need something stateful, which is why using an object is appropriate.

The data will be a string, that is, a sequence of characters (Dafny defines **`string`** as a synonym for