1 Arrays

The **Array** class is very commonly used in Ruby programs. Much like lists in SML, Ruby arrays are a standard choice for representing a collection of objects. Compared to arrays in Java (and C, C++, C#), Ruby arrays are more flexible but can less efficient.

An array is a mapping from numbers (the *indices*) to objects. Expression `[e1, e2, e3, e4]` creates a new array with four objects in it: the result of evaluating `e1` is at index 0, the result of `e2` is at index 1, etc. Note that indexing starts at 0.

There are other ways to create arrays. For example, `Array.new(x)` creates an array of length `x` with each index initially mapped to `nil`.

The syntax for getting and setting array elements is similar to many other programming languages. Expression `a[i]` gets the element at index `i` of the array referred to by `a` and `a[i] = e` sets that element to the result of `e`. Since Ruby is so object-oriented, these expression forms are really just syntactic sugar for calling methods of object `a`. In fact, `a[i]` is equivalent to `a.[i]`, and `a[i] = e` is equivalent to `a.[i]=(i, e)`.

There are many ways Ruby arrays behave differently than you might expect:

- An array can hold objects that are instances of different classes, e.g. `[14, "hi", false, 34]`.
- Negative array indices are allowed and interpreted from the end of the array. So `a[-1]` retrieves `a`'s last element, `a[-2]` retrieves the second-to-last element, etc.
- There are no array-bounds errors. If `a` holds fewer than `i+1` objects, then expression `a[i]` will evaluate to `nil`. And expression `a[i]=e` will cause `a` to dynamically grow dynamically to hold `i+1` objects—the last of which will be the result of `e`. In between the old last element and the new last elements, the indices will map to `nil`.

There are many methods and operations defined in the standard library for arrays. If the operation you need to perform on an array is at all general-purpose, it is surely already provided. As two examples, the `+` operator is defined on arrays to mean concatenation (a new array where all of the left-operand elements precede all of the right-operand elements), and the `|` operator is like the `+` operator except it removes all duplicate elements from the result.

**Arrays implementing other data structures.** In addition to all the conventional uses for arrays, Ruby arrays are also often used where in other languages we would use other constructs for tuples, stacks, or queues.

Tuples are the most straightforward usage. After all, given dynamic typing and less concern for efficiency, there is little reason to have separate language constructs for tuples and arrays. For example, for a triple, just use a 3-element array.

For stacks, the **Array** class defines convenient methods `push` and `pop`. The former takes an argument, grows the array by one index, and places the argument at the new last index. The latter shrinks the array by one index and returns the element that was at the old last index. Together, this is exactly the last-in-first-out (LIFO) behavior that defines the behavior of a stack.

For queues, the `push` method enqueues elements as just described. The `shift` method dequeues elements by returning the object at index 0 of the array, after removing it from the array and shifting all other elements left one index—that is, the object (if any) previously at index 1 is now at index 0, etc. Though not needed
for simple queues, Array also has an `unshift` method that is like `push` except it puts the new object at index 0 and moves all other objects right by 1 index, thus growing the array size by 1.

2 Hashes

Hash is a standard-library class that is quite useful in day-to-day programming. Hashes support many of the same iterator methods as arrays, which reinforces the idea that “how to iterate” is a separate problem from “what to do at each iteration.”

A hash is like an array, except that it maps from `objects` to objects instead of `numeric indices` to objects. Hashes map `keys` to `values`. All the keys in a hash must be distinct. Hashes can be created with expressions like

```
h = {"SML" => 1990, "Smalltalk" => 1980, "Ruby" => 1995}
```

We can get and set values in a hash using the same syntax as for arrays:

```
h["a"] = "xyzzy"
h[false] = 0
h["a"]
h[false]
```

Note that the objects used as keys don’t need to all be instances of the same class, and the same is true for values.

Useful methods defined for Hash include `keys` (return an array of all keys), `values` (similar, for values), and `delete` (given a key, remove it and its value from the hash). Hashes also support many of the same iterators as arrays, such as `each` and `reduce`. But some of those iterators take both keys and values as arguments, so consult the documentation.

3 Blocks

Although Ruby has `while` loops and `for` loops, stylish Ruby code typically avoids them. Instead, every method optionally takes an extra argument called a `block`. Blocks are almost, but not quite, the same thing as function closures in ML. Blocks can be used to express many of the computations you might otherwise express with loops.

For example, integers have a `times` method that takes a block and executes it the appropriate number of times. For example,

```
3.times { puts "hi" }
```

prints "hi" three times. To pass a block to a method, you put it in braces after the method call. The example above has no regular arguments, but a method can take any number of regular arguments and then 0 or 1 block.

Blocks are closures in the sense that they can refer to variables in scope where the block is defined. For example, after this program executes, `y` is bound to 3:

```
y = 0
[4,6,8].each { y += 1 }
```
Here \([4,6,8]\) is an array with three elements. Arrays have a method `each` that takes a block and executes it once for each element.

Typically, however, we want the block to actually use each array element. For example, here’s one way to sum an array’s elements and print out the running sum at each point:

```ruby
sum = 0
[4,6,8].each { |x|
  sum += x
  puts sum
}
```

Each time the block is called, \(x\) will be bound to an element of the array. Block syntax is really just a way to write an anonymous function! Ruby style typically uses `do...end` instead of braces for multi-line blocks, however:

```ruby
sum = 0
[4,6,8].each do |x|
  sum += x
  puts sum
end
```

Many collections, including arrays, have a variety of methods that look very familiar to functional programmers. For example, `reduce` is essentially the same as the `foldl` we studied in ML:

```ruby
sum = [4,6,8].reduce(0) { |acc,elt| acc + elt }
```

The argument to `reduce` is the initial accumulator. If you omit it, `reduce` will use the 0th element of the array as the initial accumulator and start with the next array element. Some other useful `iterators` (methods that use block arguments) are `map` (which you already know from ML) and `any?` (which is essentially the same as ML’s `exists`). It turns out the `Array.new` method is also an iterator. For example, in

```ruby
Array.new(x) { 0 } # or just Array.new(x,0)
Array.new(5) {|i| i}
```

the first line creates an array of length \(x\) with all elements initialized to 0. The second line creates array \([0,1,2,3,4]\).

**Using blocks in your own methods.** You can also define your own methods that take blocks. A method body calls its argument block using the `yield` keyword\(^1\). For example, this code prints "hi" once then "bye" twice:

```ruby
def foo x
  if x
    yield
  else
    yield
  end
end
foo(true) { puts "hi" }
foo(false) { puts "bye" }
```

\(^1\)Why `yield`? It was the keyword used for returning values from iterators in the language CLU, invented by Barbara Liskov in the mid 1970s.
To pass arguments to a block, you put the arguments after the `yield`, e.g., `yield 7` or `yield(8,"str").

It should now be clear that, in fact, you can pass a block to any method, but the method is free to ignore the block argument by never calling `yield`. An error will result if `yield` is called but no block was passed. The behavior when the block and the `yield` disagree on the number of arguments is rather flexible and not described in full detail here.

**Blocks are second-class closures.** Blocks are not quite first-class closures because they are not objects. Blocks cannot be stored in variables, passed as normal method arguments, put in an array, etc. In ML, we could do all those things, because closures were values like any other value. However, Ruby has first-class closures, too...

4 Procs

Class `Proc` represents first-class closures, which are objects. To make a `Proc` out of a block, just write `lambda { ... }` where `{ ... }` is any block. It turns out that `lambda` is not a keyword; it is a method of class `Object` (and every class is a subclass of `Object`, so `lambda` is available everywhere). Method `lambda` just creates a `Proc` out of the block it is passed.

To apply a closure `x` to arguments, use the closure’s `call` method—for example, `x.call` (for no arguments) or `x.call(3,4)` (for two arguments).

Rubys design is an interesting contrast from ML, which provided only first-class closures. Is it better to distinguish blocks from procs? Or would it be better just to have one, fully-powerful feature? Ruby programmers do seem to enjoy the “lightweight” syntax of blocks.