Introduction to Patterns: The Iterator

Notes

Design Patterns are well-established solutions to standard problems.

In this instance the problem is that of traversing a collection in order to perform some task. It is an important illustration of the principles of **decoupling** and **separation of responsibilities**, as well as the **single responsibility** principle.

> The **Iterator Pattern** allows traversal of the elements in a collection in a way that decouples the act of iteration and reliance on a particular collection implementation from the task to be performed on those elements.

Essentially the iterator pattern allows you to traverse a collection without needing to know how specifically that collection is constructed (it could be an array, a linked list, a heap, a set etc).

Conversely, the collection can offer access to its elements without needing to know anything about how those elements are to be used.

This is an important separation of responsibilities: The collection is responsible for maintaining it elements, adding new elements or removing them when appropriate, and so on. It needs to provide access to the elements in some fashion, but it does *not* need to know about how its clients will use those elements.

The iterator pattern consists of two parts:

1. The collection needs to have a method iterator that returns an "iterator object". In a statically typed language, this would be enforced with an interface.
2. The Iterator class consists of objects that have two methods:
    - hasNext: Returns a boolean telling us whether there is a next element or not.
    - next: Returns the next element (behavior undefined if hasNext has returned false).
    - The "contract", "promise", that the iterator class will make is that if hasNext returns true then calling next will return the next number in the collection. No guarantees are made in the event that you call next before calling hasNext, or if you call next when hasNext has returned false. We will see towards the end how to add a "forceContract" method. After this method is in place, it should prevent you from calling next unless it is after a successful hasNext.

Let us start implementing the iterator class. If you are following this on your own, you should try to implement each step on your own before reading the solution.

```
var Iterator = (function() {

    var Iterator, proto;
```

```
    Iterator = {};
    proto = {};
    Iterator.prototype = proto;

    // Basic constructor
    Iterator.new = function newIterator(next, hasNext) {
        var o = Object.create(proto);
        o.next = next;
        o.hasNext = hasNext;
        return o;
    };

    // Class method
    Iterator.fromArray = function(arr) {
      var i = -1;
      return Iterator.new(
         function next() { i += 1; return arr[i]; },
         function hasNext() { return i + 1 < arr.length; }
      );
    };

    return Iterator;
}());
```

This will be a very barebones implementation. Let us use it:

```
var it = Iterator.fromArray([1,2,3,4]);
var sum = 0;
while (it.hasNext()) { sum += it.next(); }
sum;
```

That's cool! We still sort of need to know we had an array to begin with, which is not create. We would need to add an "iterator" method to the Array prototype, and any other collection we want to traverse this way. But at least our main logic only cares about the "it" object, and can forget about the array. We can freely change how we represent our data, and the only code we might need to change is the definition of "it".

The basic implementation of iterators more or less ends here. However, we can enrich our iterators with many useful methods. forEach and reduce would be two very straightforward ones. These methods would go to the prototype object:

```
proto.forEach = function(f) {
    while (this.hasNext()) { f(this.next()); }
    return this;
};
proto.reduce = function(f, init) {
    while (this.hasNext()) { init = f(init, this.next()); }
    return init;
};
```

So now we can do:

```
var it = Iterator.fromArray([1,2,3,4]);
it.reduce(function(acc, v) { return acc + v; }, 0);
it = Iterator.fromArray([1,2,3,4]);
var sum = 0; it.forEach(function(v) { sum += v; }); sum;
```

Now let us discuss two trickier methods: map and filter. These need to return iterators that transform/skip values.

```
proto.map = function(f) {
    var that = this;
    return Iterator.new(
        next: function() { return f(that.next()); },
        hasNext: function() { return that.hasNext(); }
    );
};
proto.filter = function(pred) {
    var that = this;
    var nextValue;
    return Iterator.new(
        function next() { return nextValue; },
        function hasNext() {
            while (that.hasNext()) {
                nextValue = that.next();
                if (pred(nextValue)) { return true; }
            }
            return false;
        }
    );
};
```

Let's take these for a spin:

```
it = Iterator.fromArray([1,2,3,4,5,6])
        .filter(function(v) { return v % 2 == 1; })
        .map(function(v) { return v * v; });
while (it.hasNext()) { console.log(it.next()); }
```

Hm it should would be nice to have some way to be able to just get an array of the results. Sounds like our prototype should have a toArray method.

```
proto.toArray = function() {
    var arr = [];
    while (this.hasNext()) { arr.push(this.next()); }
    return arr;
};
```

Now we can do:

```
Iterator.fromArray([1,2,3,4,5,6])
    .filter(function(v) { return v % 2 == 1; })
    .map(function(v) { return v * v; })
    .toArray();
```

Let us now add a couple more convenience methods to our prototype:

- find: Return the first element that matches the predicate
- any: True if there is an element that passes the predicate
- every: True if every element passes the predicate
- takeUntil: Return iterable that stops on the first element that fails the predicate
- take: Return an iterable that only keeps the first n elements

- dropWhile: Return an iterable that skips all elements at first until it finds one that passes the predicate
- drop: Return a predicate that skips the first n elements
- concat: Takes any number of iterators. Returns an iterator that "concatenates them", moving to the second iterator when the first is done and so on.
- alternate: Given any number of iterators, returns an iterator that returns one element from each iterator until they are all exhausted.
- repeat: Repeats an iterator indefinitely
- partition: Given predicate, returns a pair of iterators for the true/false values respectively.
- map2: Given a second iterator and a function as argument, applies the function to pairs of values, one from each iterator. Some times called zipWith.
- cummulative: Given a function and a start value, returns an iterator that uses the function to accumulate the previous result with the next value to produce its next value.

Finally, we can add some class methods to allow us to construct some standard iterators.

- sequence: Creates a sequence. Different forms of calling it:
  - Iterator.sequence(from, to, step)
  - Iterator.sequence(from, to): step = 1
  - Iterator.sequence(from): to = Infinity, step = 1
  - Iterator.sequence(): from = 1, to = Infinity, step = 1
- constant: Given a value, creates an iterator that keeps returning that value.
- fromFunction: Given a function, creates an iterator that keeps calling that function.

For instance, to create an array of length 100 filled with random numbers, we could do:

```
Iterator
    .constant(0)
    .map(Math.random)
    .take(100)
    .toArray();
// Or simpler:
Iterator
    .fromFunction(Math.random)
    .take(100)
    .toArray();
```

Look at the complete file[1] for the implementation of these methods after you try to do them on our own.

Before we close, we will discuss the idea of "enforcing the contract". We want to give clients of our iterator the option to "force" the iteration process, so that a next

[1] testPages/iterators.js

only precedes a successful hasNext. We can do this in a prototype method: What this method does is simple:

1. Keeps a variable hasNextSuccessful that is true if hasNext has been called and was positive, and a next has not been called yet.
2. Keeps variables oldNext and oldHasNext that hold the actual next and hasNext methods.
3. Implements its own next and hasNext to do the appropriate checks before calling the old methods, and replaces the iterator's methods with these.
4. Returns the iterator.

Let's see how this might look:

```
proto.forceContract = function() {
    var hasNextSuccessful = false,
        oldNext = this.next,
        oldHasNext = this.hasNext;
    this.next = function() {
        if (hasNextSuccessful) {
            hasNextSuccessful = false;
            return oldNext.call(this);
        }
        throw new Error("'Next' called without successful 'hasNext'");
    };
    this.hasNext = function() {
        hasNextSuccessful = oldHasNext.call(this);
        return hasNextSuccessful;
    };
    return this;
};
```