Chapter 9

Higher-Order Sequential Operations

Many of the operations we wish to perform over lists have common structure. In this chapter, we investigate the most common of these patterns and how we can abstract them into powerful, generic list manipulation algorithms.

9.1 Iteration

The most common operation we perform over a List is the *traversal*. We walk the list, doing something to each element, *e.g.*, finding the largest element, summing up all the elements, or transforming all the elements. To do this, we normally write a for-loop:

```
List<T> list = /* ... */;
for (int i = 0; i < list.size(); i++) {
    ... list.get(i) ...
}
```

If our list is an `ArrayList<T>`, then this loop has linear time complexity because we perform a constant-time operation, `get`, on each element of the list. However, if the list is `LinkedList<T>`, then this loop has quadratic time complexity because `get` now takes linear time.

This additional overhead for linked list traversal seems unnecessary, however. We know that we can implement such a traversal in linear time by performing a standard linked list traversal and holding onto the *current node* rather than the *current index*. However, the `Node` class is inaccessible outside of the `LinkedList` class because we designed the `Node` class to be `private` and nested within the `LinkedList` class[1] We need some mechanism for *abstracting* the method by which we traverse a particular list in an efficient manner, whether that list is an array list or linked list. This mechanism is called *iteration* and in Java we use an `Iterator` object to perform the traversal.

We can think of an iterator as a *cursor* into our list, maintaining our current position as we iterate over the structure. Initially, the iterator points to the beginning of the list, and we can fetch the element that the cursor currently references. The `List` interface of the Java standard library provides a method, `iterator()`, that produces such an iterator object. Here is an example of using an iterator object to walk a `List`:

[1]It is important to note that hiding the `Node` class from clients of the `LinkedList` class is not a hard-and-fast rule; it is a design choice. Some libraries such as the .NET library (the standard library of the C# programming language) expose its `Node` class to clients, trading off uniformity and simplicity of interface for flexibility in manipulating the structure of the linked list.

```
List<String> list = /* ... */;
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

The `Iterator` itself is an interface with two critical operations:

- `boolean hasNext()`: returns `true` if the cursor is pointing to an element of the list, *i.e.*, there are still elements left to process.
- `T next()`: retrieves the element currently under the cursor *and* advances the cursor one position.

The `ArrayList` and `LinkedList` provide two implementations of this interface:[2]

- The array list's iterator object maintains the current index as its cursor. `hasNext` checks to see if the index exceeds the list's size. `next` retrives the element at the current index and increments the index afterwards.
- The linked list's iterator object maintains a pointer to the current node as its curosr. `hasNext` checks to see if the pointer is pointing to `null`. `next` retrives the element stored in the node currently being pointed to and advancing the pointer afterwards.

Because of this, the list iteration code above using an `iterator` operates in linear time irrespective of the type of list being used!

Trade-offs By using the `Iterator` class, we can write code that efficiently traverses a list that operates uniformly over both array lists and linked lists. In addition to this solution being more concise than a traditional for-loop, less can "go wrong" with this code. There are no indices or references to worry about—indeed, the iterator object provides exactly the set of operations we need to perform a forward traversal and nothing more. However, in return, note that the iterator interface constrains us to only move forward through a list. In particular, we cannot:

- Go backwards in the list.
- Go through two lists *pairwise*, *i.e.*, look at the first elements from each list together, then the second elements, and so forth.
- Delete elements from the list.

While other kinds of iterators that Java provide backwards movement (*e.g.*, the `ListIterator` class), in general we cannot capture all of the patterns of iteration or behavior we might want to perform. However, basic iteration is so common when using sequential structures (indeed, any data structure), that the behavior that an iterator provides is still useful even though it is highly specialized.

[2] The Java `Iterator` interface specifies a third, optional method, `remove()`, which removes the element currently being pointed at by the iterator. Removal is considered optional because some data structures may not support removal. Java's list data structures (`ArrayList` and `LinkedList`) support `remove`, but we might want to iterate over something whose elements cannot be directly modified, *e.g.*, an immutable list. In this case, `remove` should throw an `UnsupportedOperationException`.

9.1.1 Iterator Implementation

How is the `Iterator` interface implemented? As described above, the `ArrayList` and `LinkedList` classes each provide their own version of the `Iterator` interface. We could perform the iteration using methods of the `List` interface, but as we noted at the top of this reading, this makes linked list traversal quadratic instead of linear. Therefore, we must implement the `Iterator` interface differently for `ArrayList` and `LinkedList`.

Array Lists To see how to implement the `Iterator` interface for each of `ArrayList` and `LinkedList`, it is useful to appeal to the code for traversing these structures. For example, recall that an array list maintains two fields—`size` and `data`. Using these fields, we can write code to traverse the elements of the list as follows:

```
int index = 0;
while (index < size) {
    ... data[index] ...
    index++;
}
```

(Note that we iterate up to the size of the list rather than the length of the array because the elements of the list are stored at indices `0` to `(size - 1)`.) To translate this traversal operation into a class, we must recognize the relevant *state* of the operation and how the operation maps onto the iterator interface's methods, `hasNext` and `next`.

For array list traversal, the state of its traversal is the `index` being incremented in the loop. In terms of this state, `hasNext` is equivalent to the guard in the for-loop and `next` is a combination of the array access inside of the loop body and the increment of `index`. Thus, we can implement the `Iterator` interface for the `ArrayList` class as follows:

```
public class ArrayList<T> {
    private int size;
    private Object[] data;
    // ...
    private class ArrayListIterator implements Iterator<T> {
        private int index;
        public ArrayListIterator() {
            index = 0;
        }
        public boolean hasNext() { return index < size; }
        public T next() { return data[index++]; }
    }

    public Iterator<T> iterator() {
        return new ArrayListIterator();
    }
}
```

The `ArrayListIterator` class implements the array list traversal-specific behavior described above. With this class, we make two important design decisions which make it an air-tight abstraction:

- It is contained within the `ArrayList` class so that it has access to an array list object's fields.
- It is marked `private` because clients of the array list class should not be aware of the `ArrayListIterator` class.

The fact that the `ArrayListIterator` is hidden from clients seems counter-intuitive. The whole point of the iterator is to give clients a way to traverse the data structure!

The way we expose this class to users is through the `iterator()` method of the `ArrayList` class which creates an `ArrayListIterator` and immediately returns it to the client. Note that the `iterator()` method returns a value of the more general type `Iterator`, rather than the actual type of the value, `ArrayListIterator`. This is sound because `ArrayListIterator` implements `Iterator`, so it provides all of the behavior expected of any `Iterator` object. But by using this particular arrangement, we hide the implementation details of the array list class—namely that it implements its iteration behavior with the `ArrayListIterator` class—from clients but still expose an iterator for external use.

Linked Lists Recall that the `LinkedList` class contains a field `first` that is a reference to the first `Node` in the linked list. The `Node` class itself has a `value`—an element of the list—and a `next` field—a reference to the next node in the list. With this in mind, we can perform linked list traversal using the following code skeleton:

```
Node<T> cur = first;
while (cur != null) {
    ... cur.value ...
    cur = cur.next;
}
```

Unlike the array list, we must keep track of the current *node* that we are on and advance the iterator by advancing our "current" reference. Otherwise, we'll use the same set up of an inner iterator class exposed with an `iterator()` method that produces an instance of this class and gives it to the user:

```
public class LinkedList<T> {
    private static class Node<T> { /* ... */ }
    private Node<T> first;
    // ...
    private class LinkedListIterator implements Iterator<T> {
        private Node<T> cur;
        public LinkedListIterator() { cur = first; }
        public boolean hasNext() { cur != null; }
        public T next() {
            T ret = cur.value;
            cur = cur.next;
            return ret;
        }
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }
}
```

Note that the `next()` method of the `LinkedListIterator` class operates in constant ($\mathcal{O}(1)$) time. This allows us to iterate over a linked list in linear ($\mathcal{O}(n)$) time similarly to an array list.

9.1.2 For-each Loops

Because traversal with an iterator is so common in Java (and in general, programming), the language provides special syntax for this through the *for-each* loop:

```
List<String> list = /* ... */;
for (String str : list) {
    ... str ...
}
```

A for-each loop is similar in syntax to a for-loop. However, a for-each loop only has two components inside of the parenthesis:

- The variable that each element of the list is bound to while the loop runs (`str` in the above example).
- The actual list we are traversing over (`list` in the above example).

The above for-each loop is equivalent to the following while-loop:

```
List<String> list = /* ... */;
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    ... str ...
}
```

In fact, when you use a for-each loop, the Java compiler replaces it with this while-loop! Such a programming language construct that acts as "shorthand" for some other combination of programming language constructs (rather than doing that cannot otherwise be replicated in the language) is called *syntactic sugar*). Thus, we say that a for-each loop is syntactic sugar for using the `Iterator` class to perform a traversal.

This behavior is not specific to lists; we can use a for-each loop to iterate over antything produces an `Iterator` object! The Java compiler checks for this by seeing if a class implements the `Iterable` interface. The `Iterable` interface requires one method of its implementors:

- `Iterator<T> iterator()`: returns an iterator to the "beginning" of this object.

This is precisely the method we use from the `List` interface to grab our iterators. If you want your own classes to be usable in a for-each loop, then you must:

- Create a structure-specific class that implements the `Iterator` interface with appropriate traversal behavior for that data structure.
- Have the data structure class implement the `Iterable` interface. The `iterator()` method you define should return an instance of the class's corresponding `Iterator` sub-class.

As a final note, it is worthwhile to realize that iteration is a general concept that applies to any sort of data structure we may encounter, not just lists. If we think of our data structures as *containers* for data, we will frequently want to access all of the data stored in a particular container. While sequential structures provide a very natural implementation of iteration based on the order of their elements, we can extend this notion to the hierarchical and mapping structures that we will encounter later.

9.2 Infinite Sequences

So far, we have studied sequential structures of *fixed size*—arrays—and *variable size*—lists. Note that these variable-sized lists, while possessing the capacity to grow, are ultimately *finite* in length. That is, the size of the list is always some finite value although it can grow as necessary. Does it make sense to study sequences with a potentially *infinite* number of values? At first glance, such a structure seems unusable! In the previous section, we identified that the most common operation over a sequence is the traversal, but how can we traverse a structure with an infinite number of values?

Indeed, this is impossible to do as such as a traversal would never end; the best we can do is sample finite prefixes of the infinite sequence What we'll do instead is rather than traversing the structure repeatedly to perform operations over it, we'll build up a collection of computations that will be performed only when we ask for elements from the stream. This *lazy model of computation* where our transformations over the data do not occur until it is necessary is not just useful for infinite sequences, they are also useful for finite sequences as well.

9.2.1 Streams

A stream is a potentially infinite sequence of values. For example, network traffic from the Internet or input from the user can be thought of as a stream because we do not know if the data from the source (the Internet or the user) will ever end. Finite sequences can naturally be thought of as a stream as well, *i.e.*, lists or files. In particular, very, very large sequences of data, say in the gigabytes or terabytes, are effectively infinite streams as we cannot practically traverse them using standard list operations.

In Java 8, the `stream` package contains a number of classes to create and manipulate streams. In particular, the `Stream` class represents this abstract data type of an infinite sequence. As a example, suppose that we wish to read in data from a file as a `Stream` object. To represent this data as a `Stream`, we have two choices: (1) read in the data into a list and then make a stream out of the list or (2) read in the data as a stream directly.

The first approach should be familiar to you. In Java, we use the `Scanner` class to read in a file line by line, word by word, *etc.*, and then place that data into a list. From there, we can use the `stream()` method of the `List` class to create a `Stream` object that reads from the list.

```java
// throws FileNotFoundException
Scanner src = new Scanner(new File("data.txt"));
List<String> data = new LinkedList<>();
while (src.hasNextLine()) {
    // Assuming that one datum appears per line in the file...
    data.add(src.nextLine());
}
Stream<String> stream = data.stream();
```

Note that the `Stream` class, like the `List`, is generic in the carrier type of the `Stream`.

The alternative approach uses additional Java 8 functionality to perform this line-by-line reading concisely:

```java
// throws IOException
Stream stream = Files.lines(Paths.get("data.txt"));
```

The `Files` class exposes a static method `lines(path)` that creates a stream to a file that will process it line-by-line. We must pass the method a relative path to the file in question represented by a `Path` object. We create such a `Path` through the static `get` method of the `Paths` class. While the first method of creating a `Stream` is more familar, it also partially defeats the purpose of using a `Stream` in the first place. By creating a `List` first, we must read the entire contents of the file into memory upfront whereas if we create the `Stream` directly, we do not incur this upfront cost.

9.2.2 Stream Processing with Higher-order Functions

With a stream in hand, we do not directly traverse it to perform transformations and analyses over the data. We instead build up a *pipeline* of operations to perform to the stream. These operations are broken up into three sorts corresponding to three methods of the `Stream` class:

- `Stream<U> map(Function<T, U> f)`: transforms the elements of the stream by applying the function `f` to each element.
- `Stream<T> filter(Function<T, Boolean> f)`: filters the stream, keeping all the elements of the stream that produce `true` when given to the function `f`
- `U reduce(U init, BiFunction<U, T, U> f)`: also called *fold*, reduces the stream to a single value by starting with the given initial value and then applying `f` to each element, accumulating a final result in the proces.

As an initial example, we can count the number of occurrences of strings in our stream the start with an "h", ignoring case with:

```
stream.map(s -> s.toLowerCase())
  .filter(s -> s.startsWith("h"))
  .count()
```

The call to `map` makes all the strings lower case. The call to `filter`, removes all strings that do not start with "h". Note that the `map` occurs first in our method chain, guaranteeing that all the strings are lower case. Finally, we then count the number of strings that are left using the `count` method which is a special case of `reduce`.

Anonymous Functions

The arguments to `map`, `filter`, and `reduce` are all functions. As a result, we call these three methods *higher-order functions* because they are functions that take other functions as arguments. In Java 8, we create these function-arguments using *anonymous function values*, also called *lambdas*.

For example, in the following call to `map` from above:

```
stream.map(s -> s.toLowerCase())
```

The single argument to `map` is the lambda `s -> s.toLowerCase()`. The lambda is a function consisting of a single argument called `s`. The body of the lambda is an expression that is evaluated and the resulting value returned when the lambda is called; here, the function produces the lower case version of its argument through the `String` class's `toLowerCase()` method. This lambda behaves similarly to this named static method declaration `f`:

```
public static String f(String s) {
    return s.toLowerCase();
}
```

As a convenience, Java will also *infer* the types of the arguments and the return type of the lambda so that you don't have to write them down.

Stream Operations

Surprisingly, we can decompose any operation we'd like to perform over a sequential structure as a combination of maps, filters, and folds. By doing so, we gain a concise, readable description of what our operation is doing. As a case in point, consider the example from above that counts the number of words that start with a "h" in a case-insensitive manner:

```
int count = stream.map(s -> s.toLowerCase())
  .filter(s -> s.startsWith("h"))
  .count();
```

versus a more traditional approach with loops:

```
int count = 0;
for (String s : list) {
    if (s.toLowerCase().startsWith("h")) {
        count += 1;
    }
}
```

It is immediately clear what the first code snippet is doing whereas we must derive that same meaning by reasoning about the various control structures in the second code snippet.

However, decomposing problems using map, filter, and reduce is not natural at first, especially when we have been trained to program in an imperative style in Java. Let's take a closer look at these three fundamental operations over sequences to get a better feel for how to use them on real-world problems.

Map A mapping operation takes a sequence and transforms it into a new sequence by applying a function to every element of that list. For example, this call to `map` transforms a stream of strings into the same stream but with all of the strings in lower case:

```
stream.map(s -> s.toLowerCase())
```

The mapping operation does not necessarily need to preserve the carrier type of the stream. For example, this call to `map` transforms a stream of strings into a stream of the lengths of strings:

```
stream.map(s -> s.length())
```

We also can chain together `map` calls together to perform a series of transformations. This series of `map` calls produces a stream of booleans that indicate if the string has odd length:

```
stream.map(s -> s.length()).map(n -> n % 2 == 1)
```

Note that the map method returns the same Stream object, so we can *chain together* calls to map to apply them in sequence.

When transforming data with map, we sometimes frequently wish to map the elements of a list to multiple datum. We can do so by utilizing a tuple, a fixed-size heterogeneous sequence. More modern programming provide built-in facilities for creating and extracting data from tuples. In Java, we have to create a class for a tuple, one for each size we wish to support. For example, here is a generic pair class (a tuple of two elements):

```
public class Pair<T1, T2> {
    private T1 fst;
    private T2 snd;
    public Pair(T1 fst, T2 snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public T1 getFst() { return fst; }
    public T2 getSnd() { return snd; }
}
```

With this, we can map data onto pairs. For example, perhaps we want to map strings onto whether they are odd length while also preserving the strings themselves. We can do this with the following calls to map:

```
  stream.map(s -> new Pair<>(s, s.length()))
        .map(p -> new Pair<>(p.getFst(), p.getSnd() % 2 == 1))
```

The result of this expression is a Stream<Pair<String, Boolean>> where each pair is a string and a boolean indicating if the length of the string is odd.

Filter A distinguishing feature of map is that it preserves the structure of the sequence. That is, it does not delete or add elements to the sequence; it simply transforms the elements of the sequence. To remove elements from a stream, we appeal to another operation, filter. Like map, filter takes a higher-order function as an argument. However, unlike map which takes a function from the stream's carrier type to some other type, filter takes a function from the stream's carrier type to boolean.

For example, if we have a stream of integers (a Stream<Integer>), we can keep all integers that are non-negative with the following filter expression:

```
stream.filter(n -> n >= 0)
```

Note that the function we provide to filter tells us which elements to *keep* in the stream. In other words, if the function returns true for an element, filter keeps the element; if the function returns false for an element, filter removes it.

Reduce Another distinguishing feature of map as well as filter is that it operates on each element of the stream *independently*. That is, the functions passed to map and filter cannot make decisions based on anything other than individual elements of the stream. However, we would like to be able to perform operations that "summarize" the stream by accumulating a value built up by analyzing each element of the stream. The reduce method allows us to do this by allowing us to supply a function that takes two values:

- The value accumulated so far and
- The current element of the stream

and returns an updated accumulated value. For example, we can sum up a stream of integers using the following call to `reduce`:

```
stream.reduce(0, (acc, n) -> acc + n)
```

In addition to the function value, `reduce` also takes an *initial value* for the accumulated value. In the event that the stream is empty, the initial value is returned unchanged. If we wish to sum up the number of elements in this stream, we can change the call to `reduce` as follows:

```
stream.reduce(0, (acc, n) -> acc + 1)
```

Unlike `map` and `filter`, `reduce` is a *consuming* operation over the stream. Because it necessarily traverses the elements of the stream, it is not lazily evaluated like `map` and `filter`. Furthermore, unlike `map` and `filter`, applying a consuming operation over a stream "uses it up" so that no further operations can be applied to that particular stream.

Finally, the presented `reduce` method is the simple version where the type of the accumulated value is the same as the carrier type of the stream. To accumulate a value of a type different from the `Stream`, you must use the following three argument version of `reduce`:

```
  reduce( /* initial accumulated value */
        , /* binary function: accumulated value, element -> new accumulated value */
        , /* binary function: two accumulated values -> new accumulated value */ )
```

The third argument, the *combiner function*, allows `reduce` to operate over different parts of the stream and then combine sub-accumulated results into a final solution[3] As a convenience, the `Stream` class also provides two additional methods that specialize the behavior of `reduce` to some common cases:

- `count()`: returns the number of elements in the stream.
- `forEach(action)`: applies the `action` function—which takes an element of the list and returns no value—to each element of the stream.

9.2.3 Functional Interfaces*

If a lambda is indeed a value, then it must have a type. What is the type of a lambda? In Java, the answer is (unfortunately) much more complicated than you might expect.

Before lambda expressions and Java 8, developers specified higher-order functions through classes. For example, the `Thread` class found in the `java.lang` package represents a thread of execution in our program. We can use threads to specify multiple threads of executions to run concurrently—*i.e.*, at the same time—in our program. The primary `Thread` constructor takes a `Runnable` object as input. `Runnable` is an interface with the following simple definition:

```
public interface Runnable {
    public void run();
}
```

[3]This implies that the binary function is an associative operation because the order that the elements are traversed is unspecified.

The single method `run()` is the action that the created `Thread` will execute.

To create a `Thread`, we must therefore create a class that implements `Runnable` to supply to our thread. Before Java 8, we had two mechanisms for doing this: create a new class whose purpose is to be supplied to our `Thread` class or use an *anonymous inner class* which allows us to define an object that implements an interface without naming a class, much like how an lambda defines an unnamed function.

As an example, here is how we might use an anonymous inner class to create a `Thread` without having to create a new class:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello World!");
    }
});
```

This thread, when run, will print "Hello World!" to the console. The syntax of an anonymous class is:

```
new <interface name>() {
    // interface implementation goes here
}
```

The result of this anonymous inner class expression is a new object that is a subtype of the given interface.

While anonymous inner classes give us significant flexibility in how we declare objects, it is overkill for situations like this where our object only needs to provide *stateless behavior*. The primary reason lambda expressions were introduced into Java 8 was to give more concise syntax for these situations. Indeed, we can use a lambda expression instead of an anonymous inner class instead in this case:

```
Thread t = new Thread(() -> {
    System.out.println("Hello World!");
});
```

By enclosing the body of the lambda in curly braces, we can specify statements like normal for its body.

For backwards compatability's sake, the designers of Java needed to give the lambda expression a type such that we can use a lambda expression in place of a `Runnable` while still also allowing old code that used anonymous inner classes that implement `Runnable` to work. Indeed, we can assign our lambda expression to a variable of type `Runnable` and it works:

```
Runnable lam = () -> {
    System.out.println("Hello World!");
};
```

However, we can also assign the lambda expression to an interface type `Action` defined as follows:

```
@FunctionalInterface
public interface Action {
    public void action();
}
// ...
Action act = () -> {
    System.out.println("Hello World!");
};
```

How does this work? Java defines a notion of a *functional interface* which is an interface that:

- Is marked with the `@FunctionalInterface` annotation in its declaration.
- Exposes a single method called its *functional method*.

The functional method defines the type signature of a particular lambda expression, *e.g.*, the `Action` functional interface above defines a type signature for lambdas that take no arguments and return no values. Any lambda that has the same type signature as a functional interface's method is considered a subtype of that functional interface. This allows the same lambda to be both assignable to a `Runnable` as well as an `Action`.

Thus, for any lambda, we can cook up an appropriate functional interface to describe its type. However, to save us that work, the `java.util.function` package defines a number of generic functional interfaces that cover most of the cases we're interested in. For example:

- `Function<T, R>` is the type of lambdas that take a `T` as input and produce an `R` as output.
- `BiFunction<T, U, R>` is the type of lambdas that take a `T` and `U` as input and produce an `R` as output.
- `Predicate<T>` is the type of lambdas that take a `T` as input and produce a boolean as output.
- `Supplier<T>` is the type of lambdas that take no arguments and produce a `T` as output.

The `Stream` class as well as other standard library methods that use higher-order functions use these pre-defined functional interfaces.