# L15: Map, Filter, Reduce

### Today

- Map/filter/reduce
- Lambda expressions
- Functional objects
- Higher-order functions

## Example

Suppose we're given the following problem: write a method that finds the words in the Java files in your project.

Following good practice, we break it down into several simpler steps and write a method for each one:

- find all the files in the project, by scanning recursively from the project's root folder
- restrict them to files with a particular suffix, in this case .java
- open each file and read it in line-by-line
- break each line into words

Writing the individual methods for these substeps, we'll find ourselves writing a lot of low-level iteration code. For example, here's what the recursive traversal of the project folder might look like:

```
    /** Find all the files in the filesystem subtree rooted at folder.
     * @param folder root of subtree.  Requires folder.isDirectory() == true.
     * @return list of all ordinary files (not folders) that have folder as
their ancestor.
     * @throws IOException if an error occurs while accessing the filesystem
     */
    public static List<File> allFilesIn(File folder) throws IOException {
        List<File> files = new ArrayList<File>();
        for (File f: folder.listFiles()) {
            if (f.isDirectory()) {
                files.addAll(allFilesIn(f));
            } else if (f.isFile()) {
                files.add(f);
            }
        }
        return files;
    }
```

And here's what the filtering method might look like, which restricts that file list down to just the Java files (imagine calling this like onlyFilesWithSuffix(files, ".java")):

```
    /** Filter a list of files to those that end with suffix.
     * @param files list of files (all non-null)
     * @param suffix string to test
     * @return a new list consisting of only those files whose names end with
suffix
     */
    public static List<File> onlyFilesWithSuffix(List<File> files, String
suffix) {
        List<File> result = new ArrayList<File>();
```

```
        for (File f : files) {
            if (f.getName().endsWith(suffix)) {
                result.add(f);
            }
        }
        return result;
    }
```

Today we're going to talk about map/filter/reduce, a design pattern that substantially simplifies the implementation of functions that operate over sequences of elements. In this example, we'll have lots of sequences – lists of files; input streams that are sequences of lines; lines that are sequences of words; frequency tables that are sequences of (word, count) pairs. Map/filter/reduce will enable us to operate on those sequences with *no* explicit control flow – not a single for loop or if statement.

Along the way, we'll also see an important Big Idea: functions as "first-class" data values, meaning that they can be stored in variables, passed as arguments to functions, and created dynamically like other values. This is unfortunately not well-realized in Java, as we'll see. It's possible to write first-class functions in Java, and useful at times (we've already done it!), but it's verbose, so it won't give us the extra simplicity that we want for map/filter/reduce. So to demonstrate the power of map/filter/reduce, we'll switch back to Python.

## Abstracting Out Control Flow

We've already seen two design patterns that abstract away from the details of iterating over a data structure: Iterator and Visitor.

Iterator gives you a sequence of elements from a data structure, without you having to worry about whether the data structure is a set or a token stream or a list or an array – the Iterator looks the same no matter what the data structure is. Visitor abstracts the details of traversal of a recursive data type.

The map/filter/reduce patterns we'll be looking at today do something similar to Iterator and Visitor, but at an even higher level – they treat the entire sequence of elements as a unit, so that the programmer doesn't have to name and work with the elements individually. In this paradigm, the control statements disappear: specifically, the for statements, the if statements, and the return statements in the code above will be gone. We'll also be able to get rid of most of the temporary names (i.e., the local variables files, f, and result).

# Map

Let's imagine an abstract datatype Seq<E>, which represents a sequence of elements ∈ E

e.g. [1,2,3,4] ∈ Seq<Integer>

Any datatype that has an Iterator can qualify as a sequence: e.g., array, list, set. A string is also a sequence, of characters, although Java's strings don't offer an Iterator. Python is more consistent in this respect. Not only are lists iterable, but so are strings, tuples (which are immutable lists), and even input streams (which produce a sequence of lines). Python's syntax is also more compact, so we'll be using it for code examples for the next few sections.

We'll have three operations for sequences: map, filter, and reduce. Let's look at each one in turn, and then look at how they work together.

**Map** applies a unary function to each element and returns a new list containing the results, in the same order.

map : (E → F) x Seq<E> → Seq<F>

(in Python)

```
from math import sqrt
map(sqrt, [1,4,9,16]) # ==> [1.0, 2.0, 3.0, 4.0]
map(str.lower, ['A', 'b', 'C']) # ==> ['a', 'b', 'c']
```

Map is straightforward to implement in Python:

```
def map(f, seq):
    result = []
    for x in seq:
        result.append(f(x))
    return result
```

This operation captures a common pattern for operating over sequences: doing the same thing to each element of the sequence.

# Functions as Values

Let's pause here for a second, because we're doing something unusual with functions here. The map function takes a reference to a *function* as its first argument – not to the result of that function. When we wrote:

```
map(sqrt, [1,4,9,16])
```

we didn't *call* sqrt (like sqrt(25) is a call); instead we just used its name. In Python, the name of a function is a reference to an object representing that function. You can assign that object to another variable if you like, and it still behaves like sqrt:

```
mysqrt = sqrt
mysqrt(25) # ==> 5.0
```

You can also pass a reference to the function object as a parameter to another function; that's what we're doing with map here. You can use function objects the same way you would use any other data value in Python (like numbers or strings or objects).

Functions are **first-class** in Python, meaning that they can be assigned to variables, passed as parameters, used as return values, and stored in data structures. First-class functions are a very powerful programming idea. The first practical programming language that used them was Lisp, invented by John McCarthy at MIT. But the idea of programming with functions as first-class values actually predates computers, tracing back to Alonzo Church's lambda calculus. The lambda calculus

used the Greek letter λ to define new functions; this term stuck, and you'll see it as a keyword not only in Lisp and its descendants, but also in Python.

We've seen how to use built-in library functions as first-class values; how do we make our own? One way is using a familiar function definition, which gives the function a name:

```
def powerOfTwo(k):
    return 2**k

map(powerOfTwo, [1,2,3,4]) # ==> [2, 4, 8, 16]
```

When you only need the function in one place, however – which often comes up in programming with functions -- it's more convenient to use a **lambda expression**:

```
lambda k: 2**k
```

This expression represents a function of one argument (called k) that returns the value $2^k$. You can use it anywhere you would have used powerOfTwo:

```
(lambda k: 2**k) (5) # ==> 32
map(lambda k: 2**k, [1,2,3,4]) # ==> [0,1,2,3]
```

Python lambda expressions are unfortunately syntactically limited, to functions that can be written with just a return statement and nothing else (no if statements, no for loops, no local variables). But remember that's our goal with map/filter/reduce anyway, so it won't be a serious obstacle.

Guido Von Rossum, the creator of Python, has written a blog post about the design principle that led not only to first-class functions in Python, but first-class methods as well:

http://python-history.blogspot.com/2009/02/first-class-everything.html

## More Ways to Use Map

Map is useful even if you don't care about the return value of the function. When you have a sequence of mutable objects, for example, you can map a mutator operation over them:

```
map(IOBase.close, streams) # closes each stream on the list
map(Thread.join, threads)  # waits for each thread to finish
```

Some versions of map (including Python's builtin map) also support mapping functions with multiple arguments. For example, you can add two lists of numbers element-wise:

```
import operator
map(operator.add, [1,2,3], [4,5,6]) # ==> [5, 7, 9]
```

## Filter

Our next important sequence operation is **filter**, which tests each element with a unary predicate. Elements that satisfy predicate are kept; those that don't are removed. A new list is returned; filter doesn't modify its input list.

filter : (E → boolean) x Seq<E> x → Seq<E>

Python examples:

```
filter(str.isalpha, ['x', 'y', '2', '3', 'a']) # ==> ['x', 'y', 'a']

def isOdd(x): return x % 2 == 1
filter(isOdd, [1,2,3,4]) # ==> [1,3]

filter(lambda s: len(s)>0, ['abc', '', 'd']) # ==> ['abc', 'd']
```

We can define filter in a straightforward way:

```
def filter(f, seq):
    result = []
    for x in seq:
        if f(x):
            result.append(x)
    return result
```

# Reduce

Our final operator, **reduce** combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an *initial value* that initializes the reduction, and that ends up being the return value if the list is empty.

reduce : (F x E → F) x Seq<E> x F → F

reduce(f, list, init) combines the elements of the list from left to right, as follows:

$result_0$ = init

$result_1$ = f($result_0$, list[0])

$result_2$ = f($result_1$, list[1])

...

$result_n$ = f($result_{n-1}$, list[n-1])

$result_n$ is the final result for an n-element list.

Adding numbers is probably the most straightforward example:

```
reduce(operator.add, [1,2,3], 0) # ==> 6
```

There are two design choices in the reduce operation. First is how to whether to require an initial value. In Python's reduce function, the initial value is optional, and if you omit it, reduce uses the first element of the list as its initial value. So you get behavior like this instead:

$result_0$ = undefined (reduce throws an exception if the list is empty)

$result_1$ = list[0]

$result_2$ = f($result_1$, list[1])

...

$result_n$ = f($result_{n-1}$, list[n-1])

This makes it easier to use reducers like *max*, which have no well-defined initial value:

```
reduce(max, [5,8,3,1]) # ==> 8
```

The second design choice is the order in which the elements are accumulated. For associative operators like add and max it makes no difference, but for other operators it can. Python's reduce is also called **fold-left** in other programming languages, because it combines the sequence starting from the left (the first element). **Fold-right** goes in the other direction:

fold-right : (E x F → F) x Seq<E> x F → F

where fold-right(f, list, init) of an n-element list produces $result_n$ from this pattern:

$result_0$ = init

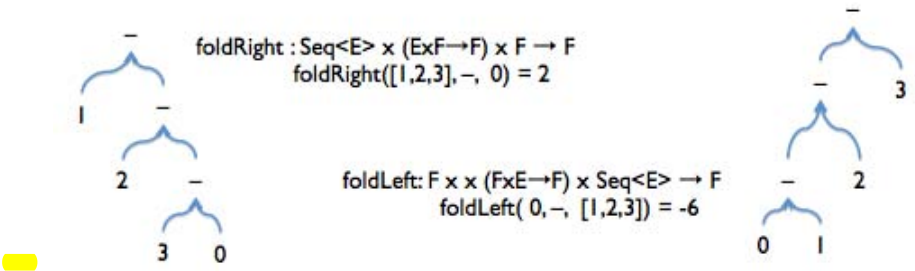
$result_1$ = f(list[n-1], $result_0$)

$result_2$ = f(list[n-2], $result_1$)

...

$result_n$ = f(list[0] , $result_{n-1}$)

**Two ways to reduce: from the left or the right**




The return type of the reduce operation doesn't have to match the type of the list elements. For example, we can use reduce to glue together a sequence into a string:

```
reduce(lambda s,x: s+str(x), [1,2,3,4], '') # ==> '1234'
```

Or to flatten out nested sublists into a single list:

```
reduce(operator.concat, [[1,2],[3,4],[],[5]], []) # ==> [1,2,3,4,5]
```

This is a useful enough sequence operation that we'll define it as **flatten**, although it's just a reduce step inside:

```
def flatten(list):
    return reduce(operator.concat, list, [])
```

# More Examples

Suppose we have a polynomial represented as a list of coefficients, a[0], a[1], ..., a[n-1], where a[i] is the coefficient of $x^i$. Then we can evaluate it using map and reduce:

```
def evaluate(a, x):
    xi = map(lambda i: x**i, range(0, len(a))) # [x^0, x^1, x^2, ..., x^n-1]
    axi = map(operator.mul, a, xi) # [a[0]*x^0, a[1]*x^1, ..., a[n-1]*x^n-1]
    return reduce(operator.add, axi, 0) # sum of axi
```

This code uses the convenient Python generator method range(a,b), which generates a list of integers from a to b-1. In map/filter/reduce programming, this kind of method replaces a for loop that indexes from a to b.

Now let's look at a typical database query example. Suppose we have a database about digital cameras, in which each object is of type Camera with observer methods for its properties (brand(), pixels(), cost(), etc.). The whole database is in a list called cameras. Then we can describe queries on this database using map/filter/reduce:

```
# What's the highest resolution Nikon sells?
reduce(max, map(Camera.pixels, filter(lambda c: c.brand() == "Nikon",
cameras)))
```

Relational databases use the map/filter/reduce paradigm (where it's called project/select/aggregate). SQL (Structured Query Language) is the de facto standard language for querying relational databases. A typical SQL query looks like this:

select max(pixels) from cameras where brand = "Nikon"

cameras is a **sequence** (a list of rows, where each row has the data for one camera)

where brand="Nikon" is a **filter**

pixels is a **map** (extracting just the pixels field from the row)

max is a **reduce**

## Finishing the Example

Going back to the example we started the lecture with, where we want to find the Java files in the project, let's try creating a useful abstraction for filtering:

```
def fileEndsWith(suffix):
    return lambda file: file.getName().endsWith(suffix)
```

fileEndsWith returns functions that are useful as filters; it takes a filename suffix like ".java" and dynamically generates a function that you can use with filter to test for that suffix:

```
filter(fileEndsWith(".java"), files)
```

fileEndsWith is a different kind of beast than our usual functions. It's a **higher-order function**, meaning that it's a function that takes another function as an argument, or returns another function as its result. Higher-order functions are operations on the datatype of functions; in this case, fileEndsWith is a producer of functions.

Now let's use map, filter, and flatten to recursively traverse the folder tree:

```
def allFilesIn(folder):
    children = folder.listFiles()
    descendents = flatten(map(allFilesIn, filter(File.isDirectory, children)))
    return descendents + filter(File.isFile, children)
```

The first line gets all the children of the folder, which might look like this:

```
["src/client", "src/server", "src/Main.java", ...]
```

The second line is the key bit: it filters the children for just the subfolders, and then recursively maps allFilesIn against this list of subfolders! The result might look like this:

```
[["src/client/MyClient.java"], ["src/server/MyServer.java"], ...]
```

So we then have to flatten it to remove the nested sublists. Then we add the immediate children which are plain files (not folders), and that's our result.

We can also do the other pieces of the problem with map/filter/reduce. Once we have the list of files we want to extract words from, we're ready to load their contents. We can use map to get their pathnames as strings, open them, and then read in each file as a list of lines:

```
pathnames = map(File.getPath, files)
streams = map(open, pathnames)
lines = map(list, files)
```

This actually looks like a single map() that we want to apply three functions to, so let's pause to create another useful higher-order function: composing functions together.

```
def compose(f, g):
    """Requires that f and g are functions, f:A->B and g:B->C.
    Returns a function A->C by composing f with g."""
    return lambda x: g(f(x))
```

Now we can use a single map:

```
lines = map(compose(compose(File.getPath, open), list), files)
```

Better, since we already have three, let's design a way to compose an arbitrary chain of functions:

```
def chain(funcs):
    """Requires funcs is a list of functions [A->B, B->C, ..., Y->Z].
    Returns a fn A->Z that is the left-to-right composition of funcs."""
    return reduce(compose, funcs)
```

so that the map operation looks more like this:

```
lines = map(chain([File.getPath, open, list]), files)
```

Now we start to see the power of first-class functions – we can put functions into data structures and use operations on those data structures, like map, reduce, and filter, on the functions themselves!

Since this map will produce a list of lists of lines (one for each file), we need to flatten it to get a single line list, ignoring file boundaries:

```
allLines = flatten(map(chain([File.getPath, open, list]), files))
```

Then we split each line into words similarly:

```
words = flatten(map(str.split, lines))
```

And we're done. As promised, the control statements have disappeared.

## Benefits of Abstracting Out Control

Map/filter/reduce can often make code shorter and simpler, and allow the programmer to focus on the heart of the computation rather than on the details of loops, branches, and control flow.

By arranging our program in terms of map, filter, and reduce, and in particular using immutable datatypes and pure functions (i.e. functions that avoid mutating data) as much as possible, we've created more opportunities for safe concurrency. Maps and filters using pure functions over immutable datatypes are instantly parallelizable –invocations of the function on different elements of the sequence can be run in different threads, on different processors, even on different machines, and the result will still be the same.

## First-class Functions in Java

We've seen what first-class functions look like in Python; how does this all work in Java?

In Java, the only first-class values are primitive values (ints, booleans, characters, etc) and object references. But objects can carry functions with them, in the form of methods. So it turns out that the way to implement a first-class function, in an object-oriented programming language like Java that doesn't support first-class functions directly, is to use an object with a method representing the function.

We've actually seen this before several times already:

- The Runnable object that you pass to a Thread is a first-class function, void run().
- The KeyListener object that you register with the graphical user interface toolkit to get keyboard events is a bundle of several functions, keyPressed(KeyEvent), keyReleased(KeyEvent), etc.
- The Visitor object that we created to implement functions over recursive datatypes did indeed represent a function -- with several variants, one for each variant of the datatype.

This design pattern is called a **functional object** or functor, an object whose purpose is to represent a function.

For the sake of implementing map/filter/reduce in Java, let's generalize this notion to a generic unary function interface:

```
/**
 * A Function<T,U> represents a unary function from T to U,
 * i.e. f:T->U.
 */
public interface Function<T,U> {
    /**
     * Apply this function.
     * @param t object to apply this function to
     * @return the result of applying this function to t.
     */
    public U apply(T t);
}
```

Then we can write map() like so:

```
    /**
     * Apply a function to every element of a list.
     * @param f function to apply
     * @param list list to iterate over
     * @return [f(list[0]), f(list[1]), ..., f(list[n-1])]
     */
    public static <T,U> List<U> map(Function<T,U> f, List<T> list) {
        List<U> result = new ArrayList<U>();
        for (T t : list) {
            result.add(f.apply(t));
        }
        return result;
    }
```

And here's an example of using map():

```
    // anonymous classes like the one below are effectively lambda expressions
       Function<String,String> toLowerCase = new Function<String,String>() {
           public String apply(String s) { return s.toLowerCase(); }
       };

       map(toLowerCase, Arrays.asList(new String[] {"A", "b", "c"}));
```

Obviously verbose, and Java is not practical for functional programming. But the notion of functors is widely used, and useful, as we've seen from examples like Runnable and Visitor.

# Higher-Order Functions in Java

Map/filter/reduce are obviously higher order functions. But let's look at two others that we introduced in today's lecture: compose() and chain().

Compose() has a straightforward implementation, and in particular once you get the types of the arguments and return value right, Java's strong typing makes it pretty much impossible to get the method body wrong:

```
/**
 * Compose two functions.
 * @param f function A->B
 * @param g function B->C
 * @return new function A->C formed by composing f with g
 */
public static <A,B,C> Function<A,C> compose(final Function<A,B> f,
                                            final Function<B,C> g) {
    return new Function<A,C>() {
        public C apply(A t) { return g.apply(f.apply(t)); }
    };
}
```

It turns out that we *can't* write chain() in strongly-typed Java, except in a very restricted form in which all the functions in the chain have the identical input and output types. This is because Lists must be homogeneous – List<Function<A,A>>.

```
/**
 * Compose a chain of functions.
 * @param list list of functions A->A to compose
 * @return function A->A made by composing list[0] ... list[n-1]
 */
public static <A> Function<A,A> chain(List<Function<A,A>> list) {
    return reduce(
        list,
        new BinOp<Function<A,A>, Function<A,A>, Function<A,A>>() {
            public Function<A, A> apply(Function<A, A> t, Function<A, A> u) {
                return compose(t, u);
            }
        },
        new Function<A,A>() {
            public A apply(A t) { return t; }
        }
    );
}
```

6.005 Elements of Software Construction

Fall 2011