Map, Filter, Reduce

CMPS 252, Spring 2020

M. Bdeir

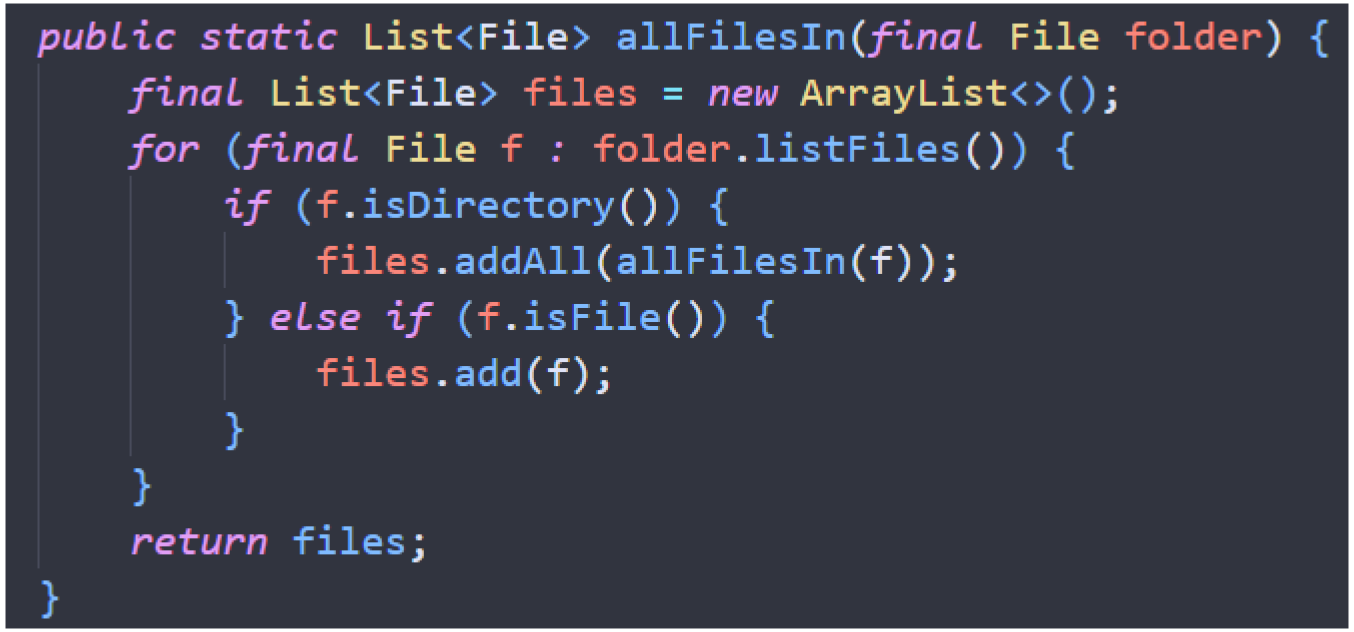
Lecture Learning Outcomes

- A design pattern for implementing functions that operate on sequences of elements
- Treating functions as *first-class values* that we can pass around and manipulate in our programs
- Map/filter/reduce
- Functional objects
- Higher-order functions

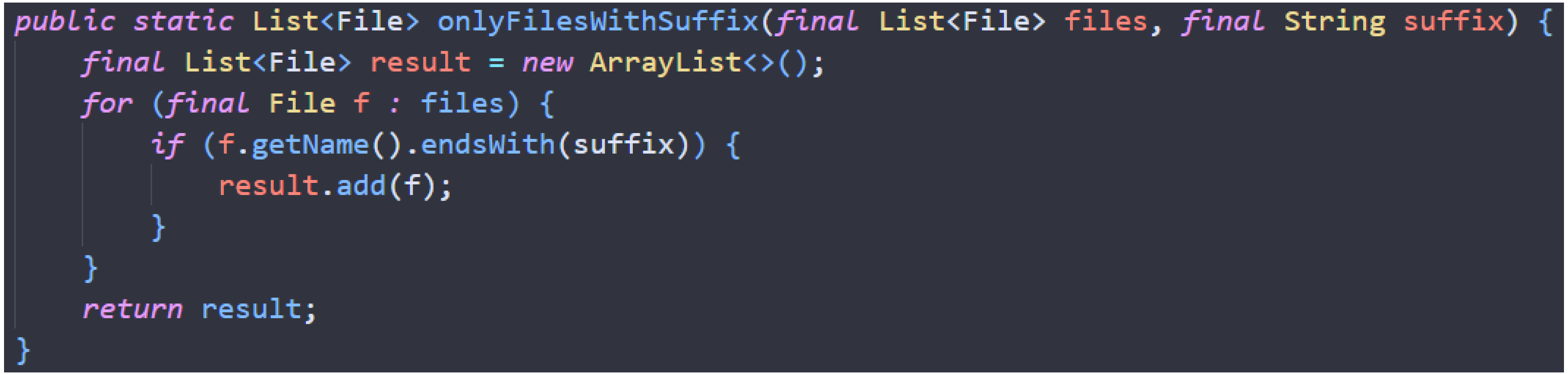
Exercise

Write a method that finds the words in the Java files in your project.

1. Find all the files in the project, by scanning recursively from the project's root folder
2. Restrict them to files with a particular suffix, in this case .Java
3. Open each file and read it in line-by-line
4. Break each line into words



```java
public static List<File> allFilesIn(final File folder) {
    final List<File> files = new ArrayList<>();
    for (final File f : folder.listFiles()) {
        if (f.isDirectory()) {
            files.addAll(allFilesIn(f));
        } else if (f.isFile()) {
            files.add(f);
        }
    }
    return files;
}
```



```java
public static List<File> onlyFilesWithSuffix(final List<File> files, final String suffix) {
    final List<File> result = new ArrayList<>();
    for (final File f : files) {
        if (f.getName().endsWith(suffix)) {
            result.add(f);
        }
    }
    return result;
}
```

Map/Filter/Reduce

- Map/filter/reduce is a design pattern that substantially simplifies the implementation of functions that operate over sequences of elements
- Examples 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 <u>(without a single for loop or if statement)</u>

Abstracting Out Control Flow

- Iterators give 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
- Iterator looks the same no matter what the data structure is

```java
String[] names = new String[] {"CMPS","MATH","CVSP"};
List<String> list = Arrays.asList(names);
```

```java
for (int i = 0; i < names.length; i++) {
    System.out.println(names[i]);
}
```

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

```java
for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);
    System.out.println(s);
}
```

```java
for(String s: new MyDatabase()) {
    System.out.println(s);
}
```

Map/Filter/Reduce Abstraction

- The map/filter/reduce patterns are similar to `Iterator` but at a higher level in that 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
- Thus, the control statements disappear (for/if/return)
- Also gone are most of the temporary names (local variables)

Map/Filter/Reduce Abstraction

- Any datatype that has an iterator can qualify as a sequence: array, list, set, etc.
- For example, [1, 2, 3, 4] ∈ Seq<Integer>
- A string is also a sequence (of characters), although Java's strings don't offer an iterator
- We'll have three operations for sequences: map, filter, and reduce

Map: A Sequence Operation

- Map applies a unary function to each element in the sequence and returns a new sequence containing the results, in the same order
- **map : (E → F) × Seq<E> → Seq<F>**

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

```
def twice(v):
    return v*2
```

```
map(twice,[1,2,3,4])
```

```
from math import sqrt
a=map(sqrt, [1, 4, 9, 16])
```

Filter: A Sequence Operation

- `Filter` tests each element of a sequence with a unary predicate returning a new list containing only those elements that satisfy the predicate
- Reminder: a predicate is a Boolean-valued function
- **`filter`: (E → boolean) × Seq<E> → Seq<E>**

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

```
def isOdd(x):
    return x % 2 == 1
```

```
filter(isOdd, [1,2,3,4])
```

Reduce: A Sequence Operation

- Reduce applies a binary function cumulatively to the elements of a sequence, optionally starting with an initial argument, and producing one value
- **reduce : (F × E → F) × Seq<E> × F → F**

```
def reduce(f,seq):
    result=0
    for elt in seq:
        result=f(result,elt)
    return result
```

```
def reduce(f,seq,start):
    result=start
    for elt in seq:
        result=f(result,elt)
    return result
```

```
def add(x,y):
    return x+y
```

```
print(reduce(add,[1,2,3,4],10))
```

```
print(reduce(add,[1,2,3,4]))
```

Design Choice for Reduce

- Using Reduce, elements can be accumulated left-to-right or right-to-left
- 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)

Benefits Of Abstracting Out Control in Map/Filter/Reduce

- Often makes the code shorter and simpler
- Allows the programmer to focus on the heart of the computation rather than on the details of loops, branches, and control flow
- More opportunities for safe concurrency (by arranging our program in terms of map, filter, and reduce, and in particular using immutable datatypes and pure functions
- 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
- MapReduce is a pattern for parallelizing large computations in this way

First-class Functions In Java

- In Java, the only first-class values are primitive values (ints, booleans, characters, etc.) and object references
- Java is an object-oriented programming language that doesn’t support first-class functions directly
- To implement a first-class function use an object with a method representing the function
- This design pattern is called a *functional object* or *functor:* an object whose purpose is to represent a function

Functional Object Or Functor in Java

- The `Runnable` object that you pass to a Thread constructor is a first-class function, void run()
- The `Comparator<T>` object that you pass to a sorted collection (e.g. `SortedSet`) is a first-class function, `int compare(T o1, T o2)`
- 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.

Functional Object Or Functor in Java

```java
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello!");
    }
}).start();
```

```java
new Thread(() -> {
    System.out.println("Hello");
}).start();
```

Standard Functional Interfaces in Java

- Java provides some standard functional interfaces we can use to write code in the map/filter/reduce pattern, e.g.:
- `Function<T,R>` represents unary functions from T to R
- `BiFunction<T,U,R>` represents binary functions from T × U to R
- `Predicate<T>` represents functions from T to boolean

Standard Functional Interfaces in Java

```java
public static <T,R> List<R> map(Function<T,R> f, List<T> list) {
    List<R> result = new ArrayList<>();
    for (T t : list) {
        result.add(f.apply(t));
    }
    return result;
}
```

```java
Function<String,String> toLowerCase = new Function<>() {
    public String apply(String s) { return s.toLowerCase(); }
};
```

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

```java
map(s -> s.toLowerCase(), Arrays.asList(new String[] {"A", "b", "C"}));
```

```java
map((s) -> s.toLowerCase(), Arrays.asList(new String[] {"A", "b", "C"}));
```

```java
map((s) -> { return s.toLowerCase(); }, Arrays.asList(new String[] {"A", "b", "C"}));
```

Map/Filter/Reduce in Java

```java
public class Words {
    static Stream<File> allFilesIn(File folder) {
        File[] children = folder.listFiles();
        Stream<File> descendants = Arrays.stream(children).filter(File::isDirectory).flatMap(Words::allFilesIn);
        return Stream.concat(descendants, Arrays.stream(children).filter(File::isFile));
    }
    static Predicate<File> endsWith(String suffix) {
        return f -> f.getPath().endsWith(suffix);
    }
    public static void main(String[] args) {
        Stream<File> result = allFilesIn(new File("C:\\Users\\CMPS 252 Spring 2020\\Modules"));
        for(var f : result.filter(endsWith(".java")).toArray()) {
            System.out.println(f);
        }
    }
}
```