Lab 8: Functional Programming in Java

CS 320 Principles of Programming Languages, Fall Term 2019
Department of Computer Science, Portland State University

Learning Objectives

Upon successful completion, students will be able to:

- Write simple Java programs making use of functional interfaces and lambda expressions for processing collections and representing characteristic functions.
- Write simple Java stream expressions.

Instructions

Download the file `lab8.zip` from D2L into any convenient directory and type

```
$ unzip lab8.zip
```

You'll see a new directory `lab8` with a set of sub-directories, 00-05X. As usual, we'll walk through many or all of these directories, in sequence. You should try compiling and running the code in each directory as appropriate. In most directories, something like `javac *.java` will be appropriate for compiling. Directories whose names end in `X` contain exercises that you will need to turn in.

When you have completed all the exercises, return to the top-level directory above `lab8` and type

```
$ zip -r sol8.zip lab8/*X*/*
```

Then submit the resulting file `sol8zip` to D2L.

For this lab you will once again want to consult the Java library API, available at `https://docs.oracle.com/en/java/javase/11/docs/api`.

For more background on the Java functional programming features used here, you may find the tutoral at `http://tutorials.jenkov.com/java-functional-programming` useful; there are many others on the web as well.

Functional Interfaces and Arrays (00)

File `Example.java` contains Java implementations corresponding to the Scala code in Lecture 8 slides 9-12. In Java (starting at version 8), we can write lambda expressions (anonymous functions) as values for any `interface` type that contains a single method. (Strictly speaking, it must be an *abstact* method, which is the default for methods in interfaces.) The name of the unique method does not matter. Such an interface is called a *functional interface*.

"00/Example.java"

```
3   @FunctionalInterface
4   interface Int2Int {
5     int apply(int x);
6   }
```

Lines 3-6 define such an interface, called `Int2Int`. The name is intended to suggest the type of its unique method, which takes and returns a single `int`. The method itself is called `apply`. The *annotation* at line 3 indicating that this is a functional interface is not required, but is helpful documentation.

"00/Example.java"

```
8   static class Add42 implements Int2Int {
9     public int apply(int z) { return z+42; }
10  }
11
12  static Int2Int add42 = w -> w + 42;
```

We can, of course, create classes that implement this interface in the normal way, as in lines 8-10. But the whole reason for focusing on functional interfaces is that we can implement them with lambda expressions, as in line 12. Here `add42` behaves exactly like an object obtained using `new Add42()`. (Note that the name of the bound variable in the lambda expression doesn't matter, any more than it does in an ordinary method definition.)

The lambda expressions we are using here are particularly simple. In general, lambda expressions can have multiple arguments, in which case these are given in parentheses and separated by commas, e.g.

```
(x,y) -> 2*x + y
```

It is sometimes necessary to declare type(s) on the arguments, which can be done like this:

```
(int x) -> x + 2
(int x,int y) -> 2*x + y
```

In order to generate an object of an appropriate class from a lambda expression, the Java compiler must know which specific functional interface is intended. It can typically figure this out from context, e.g. from the return type in the following example:

"00/Example.java"

```
14  static Int2Int adder(int x) {
15    return z -> x + z;
16  }
```

Of course, the type of the functional interface must be consistent with the lambda expression; otherwise, a static typing error occurs. (See what happens if you change the lambda expression in line 15 to `z -> "foo"` or `(double z) -> x`.)

This example also illustrates that the body of a lambda can mention variables defined in an outer scope (in this case the `x` parameter to `adder`). This is only allowed when the variables are declared `final` or the compiler can figure out that they are *effectively final*, i.e. once they are initialized they are never assigned to again.

"00/Example.java"

```
22  static void mapArray (int[] a, Int2Int f) {
23    for (int i = 0; i < a.length; i++)
24      a[i] = f.apply(a[i]);
25  }
```

Using a class implementing a functional interface is completely standard; we simply invoke the (unique) method name. Here we update each element of the argument `int` array (in place) with the result of calling the `apply` method of the `Int2Int` object `f`.

In practice, we often almost forget the difference between a functional interface object and its unique method, and just think of it as a "function." In this sense, `mapArray` takes a function `f` as argument, and `adder` returns a function as result; both are therefore simple examples of *higher-order* functions. The functions being passed and returned in this way are *first-class* functions, in the sense that they are manipulated just as easily as values of other types, such as `int` or `Object`. Unfortunately, we cannot completely ignore the fact that `f` is really an object, because we must remember and invoke its unique method name. This is one of the main reasons that functional programming in Java is klunkier than in more "functional" languages, such as Scala, Haskell, or Python, where we can simply apply first-class functions by name just like other, ordinary functions.

Note that `mapArray` is definitely *not* a pure function; it changes the contents of its array argument. Higher-order functions and lambdas are useful even in an imperative language!

Explore the code in this file and try changing various definitions to see their effect on the array.

Standard Functional Interfaces and Lists (01)

In this section we examine a simple type `IntList` of null-terminated integer linked lists and illustrate both imperative and purely functional operators on them, written as higher-order functions.

In section (00), we invented our own functional interface `Int2Int`. Usually, however, this is unnecessary; the Java standard library package `java.util.function` (see `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html`) predefines a large number of functional interfaces, suitable for many occasions. For example, our `Int2Int` interface is the same as the `IntUnaryOperator` interface. Another useful library name is `IntConsumer`, describing functions that take an `int` argument and return `void`. Starting in this section, we switch to using these library functional interface names. Unfortunately, the different library functional interfaces don't all use the same name for the unique function, so you have to look it up: for `IntUnaryOperator` the function is called `applyAsInt` and for `IntConsumer` it is called `accept`.

"01/Example.java"

```
16  static IntList map(IntList list,IntUnaryOperator f) {
17    if (list != null)
18      return new IntList(f.applyAsInt(list.value),map(list.next,f));
19    else
20      return null;
21  }
22
23  static void mapInPlace(IntList list,IntUnaryOperator f) {
24    while (list != null) {
25      list.value = f.applyAsInt(list.value);
26      list = list.next;
27    }
28  }
```

We give two versions of a `map` operator that applies an int-to-int function to each element of the list. `map` is recursive and purely functional; it returns a list of completely new links, leaving the original list unchanged. `mapInPlace` is iterative and imperative; it rewrites the values of the existing links. Make sure that you understand the difference in how these behave. Note that here the style in which the function is written

(purely functional or imperative) is consistent with the treatment of the lists (immutable or mutable). This alignment makes sense, but it is not essential; e.g. we could give a recursive definition of `mapInPlace` that does not rely on updating the variable `list`.

"01/Example.java"

```
30   static void doAll(IntList list,IntConsumer f) {
31     while (list != null) {
32       f.accept(list.value);
33       list = list.next;
34     }
35   }
```

`doAll` is an imperative operator that applies function `f` to each element of the list in order. We use it to print the contents of the list, like this:

```
IntList.doAll(list,i -> System.out.print(i + " "));
```

This does essentially the same work as an iterator (if we had defined one for `IntList`):

```
for (int i: list)
  System.out.print(i + " ");
```

Indeed, if a language has higher-order and first-class functions, perhaps is doesn't need iterators too.

Standard Functional Interfaces and Trees (02X) [10 pts]

In this section, we apply the ideas of (01) to simple binary search trees (BST's) with integers at the nodes and null pointers to represent leaves. File `Example.java` contains code that needs several holes filled and routines changed so that the driver routine (in class `Example`) will behave as the comments say it should. The basic idea of the driver is to read integers from the command line and insert them into a BST. Then we retrieve and print the elements in two different ways, which should produce identical results. Finally, we create a new tree whose elements are each twice as big as in the original tree, and print it, also making sure that the original tree has not changed.

For example, executing

```
java Example 1 3 5 9 7 5 10 -99
```

should produce the output

```
-99 1 3 5 7 9 10
-99 1 3 5 7 9 10
-198 2 6 10 14 18 20
-99 1 3 5 7 9 10
```

Currently, `Example.java` will compile, but executing `java Example` will raise an exception due to missing function bodies.

You have the following tasks:

1. Replace the body of `lookup` with a version that returns the same result as the existing one, but is purely functional, i.e. no assignments (or any other side effects) are allowed.

2. Provide a body for `map`. This should behave similarly to the `map` method of (01), except on trees. That is, it should produce a *new* map with the same shape as its argument, but with each element value x replaced by `f(`x`)` (thinking of `f` as a "function"). Again, your code must be purely functional: no assignments or other side-effects are allowed.

3. Provide a body for `inorder`. This should be behave similarly to the `doAll` method of (01), except on trees. It should visit the tree elements in "in order," so that the elements of a BST are processed in increasing sort order.

Characteristic Functions (03X) [10 pts]

The code in `Example.java` gives a Java version of the Haskell code from Lecture 8 slides 30 and 31. This code represents (infinite) sets of points (i.e. regions of the plane) as functions of type `Point -> boolean`. The function returns `true` iff it is called with a point that is in the set. Functions representing sets like this are sometimes called *characteristic functions*. They are convenient in mathematics; with the aid of functional programming tools they can be convenient for programming too.

"03X/Example.java"

```
17  class PSet {
18    Predicate<Point> ps;    // representation of a point set
19    private PSet(Predicate<Point> ps) {this.ps = ps;}
20
21    boolean in(Point p) {
22      return ps.test(p);
23    }
24
25    static PSet disk(Point center, double radius) {
26      return new PSet(point -> center.distance(point) <= radius);
27    }
28
29    PSet intersect(PSet set) {
30      return new PSet(point -> ps.test(point) && set.ps.test(point));
31    }
```

In Java, we represent point sets as objects of class `PSet` having just one field: the characteristic function. This is represented as a functional interface object `ps` of type `Predicate<Point>`. `Predicate<E>` is a parameterized functional interface whose single function is `boolean test(E e)`. The object method `in` tests a point for membership in the point set simply by invoking `ps` on it. The static method `disk` creates a set consisting of all the points in a disk (including the boundary), given the disk's center and radius. The object method `intersect` takes a second point set as argument and returns a new set which contains just those points common to both original sets (which are not changed). The `Example` driver repeats the example from slide 31, with the x and y coordinates of the test point given as the command line arguments.

The other methods in `PSet` have undefined bodies. Your task is to fill them in, given the following specifications:

- `rect` generates a new set consisting of all the points in the rectangle (including a boundary) with opposite corners given by the two arguments and with sides aligned with the x- and y-axes.
- `union` takes a second point set as argument and returns a new set which contains those points belonging to either of the original sets (which are not changed).

- `complement` returns a new set which contains all those points not belonging to the receiver (which is not changed).
- `reflectx` returns a new set which contains all the points of the receiver reflected over the x-axis. (The receiver is not changed.)

Be sure to test your code thoroughly.

Streams (04)

Java versions from 8 onwards also have extensive library support for *streams*. A stream is a sequence of values, but it is not the same as a list (or any other collection). Quoting from the Java library documentation (`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html`):

"Streams differ from collections in several ways:

- No storage. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
- Functional in nature. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Laziness-seeking. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
- Possibly unbounded. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
- Consumable. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source."

Streams are similar to the (lazy) lists in Haskell. They can be used to write compact and elegant solutions to many problems at a high level of abstraction. The code in `Example.java` illustrates three stream pipelines, and a variety of stream generators, intermediate transformers and terminal operations. The examples here all use methods of the `IntStream` interface; similar methods are available for streams of arbitrary Objects (and for `long` and `double` primitive types). The methods are documented at `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html`.

"04/Example.java"

```
7     IntStream.range(0,10)                          // generate stream of numbers 0 to 9
8       .forEach(x -> System.out.print(x + " "));  // print each one
```

Each stream pipeline starts with a stream generator. In this case we use a `range` generator from the library. This produces the stream 0,1,2,...9. Each stream pipeline ends with a terminal operation; in this case, the

terminal operation comes immediately: the `forEach` method applies an `IntConsumer` function to each element of the stream, in this case to print them out. Note that this is extremely similar to the behavior of the `RangeIterator` you wrote in Lab 7!

"04/Example.java"

```
11  int a[] = {1,3,5,5,7,3,8,9,8};
12  int sum =
13    Arrays.stream(a)          // generate stream from array
14    .map(x -> x*x)            // square each number
15    .filter(y -> y % 2 == 0) // keep only even numbers
16    .sum();                   // add up
17  System.out.println(sum);
```

The interesting part of streams usually comes in the middle: between the generation and terminal operations, we can insert intermediate operations that transform the stream. In the second example, we create a stream from the contents of array `a`. We then perform a `map` operation specified by a `IntUnaryOperation` over the stream, converting each number to its square. The `filter` operation edits the stream, keeping only elements that pass the `IntPredicate` test (here for evenness). Finally, the terminal operation `sum()` adds up the remaining elements. Note that the original array does not change; streams should never mutate their generating objects, and the standard library functions obey this.

It is important to realize that these stream expressions describe a lazy pipeline: values only flow when the terminal operation requests them from its predecessor in the pipeline, which in turn may request them from *its* predecessor, and so on until the generating method is reached. Only then are stream items actually generated.

"04/Example.java"

```
19  int c =
20    IntStream.iterate(1,x -> 2 * x + 3)   // generate the infinite stream 1, 5, 13, 29, ...
21    .takeWhile(x -> x <= 100)             // cut stream off after values exceed 100
22    .reduce(1, (x,y) -> x*y);             // compute product
23  System.out.println(c);
```

The last example illustrates use of an infinite stream. The `iterate(`s`,`f`)` method produces the stream $s, f(s), f(f(s)), ...$ where f is a `IntUnaryOperation`. To be useful, we need a way to cut off the infinite stream before we reach a terminal operation. One way is with the `takeWhile` method, which keeps just the prefix of the stream that obeys its `IntPredicate` argument. Finally, we use `reduce` to take the product of the (now finite) stream. Applying `reduce(`c`,`f`)` to to stream $x_1, x_2, \ldots, x_n$ where f is a two-argument `IntBinaryOp`, computes $f(x_1, f(x_2, \ldots, f(x_n, c)\ldots))$. It should only be used for associative operations, where c is the identity value for the operation. The built-in library operations like `sum()`, `max()`, and `count()` can all be coded as instances of `reduce()`.

Streams Exercises (05X) [10 pts]

Use stream pipelines in the style of the previous section to solve the following tasks. You can use any of the library operators for `IntStream`s. There is often more than one way to accomplish a given stream transformation. You may *not* use any imperative techniques (assignments, mutable data structures, etc.) or iterators.

These exercises assume that the command line arguments are parsed as integers into an array of values `a`.

- Find and print out the maximum value in `a`. You may assume `a` has at least one element.

- Find and print (one per line) those values of `a` that are of the form $3x + 1$ for some x. Each such value should be printed only once, in the position it first appears.
- Find and print (one per line) those values appearing between the first and second 0 values in `a`. (For example, if `a` contains 1,2,0,4,5,6,0,7,8,0,9 then you should print 4,5,and 6.) You may assume there are at least two instances of 0 in `a`.
- For each value x appearing in `a`, print out (one per line) x copies of x. (For example, if `a` contains 2,3,4 you should print out 2,2,3,3,3,4,4,4,4.) You may assume that all values in `a` are non-negative.
- Find and print out the last value in `a`. You may assume `a` has at least one element.

For example, running

```
java Example 1 0 4 2 0 4 0
```

should produce the output

```
part a:
4
part b:
1
4
part c:
4
2
part d:
1
4
4
4
4
2
2
4
4
4
4
part e:
0
```