

Lists in Java

COMP1100 — Introduction to Programming and Algorithms

Reading: *Big Java Ch.8.2 – 8.4*

Clem Baker-Finch

Australian National University

Semester 1, 2006

Java Lists

The Java API has several different kinds of lists.

(For example, `ArrayList`, `LinkedList`, `Vector`, `Stack`...)

They all have a large collection of methods in common — see the `java.util.List` interface.

So how do they *differ*?

The main difference is in terms of their *implementations*.

How do we know *which version to choose* for a particular application?

Wait until `COMP1110/1510`!

Lists

Recall that **lists** allow us to combine a **varying number** of values of **the same type**.

Lists are an extremely important data structure in programming, so we spent a lot of time working with lists in Haskell.

In Haskell, lists are **built in** so we have some nice notation like `[3, 5, 7]` and list comprehensions.

In Java, lists are supplied by the **Java Class Libraries** in the package `java.util`.

`java.util.ArrayList`

In this course we will choose to use `ArrayList` and not worry about its implementation.

Some of the basic `ArrayList` methods:

`add(elem)` an element to the **end** of this list.

`get(index)` the element in the **indexth position** in the list.

`isEmpty()` tests if the list has no elements.

`size()` returns the number of elements in the list.

and so on.

Notice that the basic Java list operations have a different style to those used to in Haskell.

In Haskell, we built lists by cons-ing elements on to the **front** of a list, like `(x:xs)`.

Java adds them to the **end**.

In Haskell we took lists apart by breaking them into their **head** and **tail**.

Java accesses list elements with the **get**.

Haskell has an equivalent operator `(!!)` but we didn't use it much.

Generic Classes

The `ArrayList` class has heading `ArrayList<E>`.

What's the `<E>` mean?

Since Java 1.5, classes may be **generic**, which is a similar concept as Haskell's **polymorphic** types.

In Haskell, if we want a list of Ints we write `[Int]`.

If we want a list of Strings we write `[String]`.

In Java, if we want a list of Integers we write `ArrayList<Integer>`.

If we want a list of Strings we write `ArrayList<String>`.

`java.util.Stack`

The `Stack` version of lists in Java does have methods more like the operations we are used to in Haskell.

peek is like **head**.

pop is like **tail**.

push is like **cons**.

It also has all the other standard Java list methods.

But Stacks are “special purpose” lists, so we won't use them here.

DrJava Demonstration ...

Traversing Lists in Haskell

In Haskell many of the list algorithms we studied were structured around *traversing* the list.

That is, the algorithm visited each element of the list (in order), processing the elements in some way.

Sometimes we used a recursive definition, e.g.

```
sum []      = 0
sum (x:xs) = x + sum xs
```

Other times we used some of the higher-order functions that represent **patterns of recursion**, like `map`, `fold`, `filter` and so on.

For-each Loop Structure

Suppose we have a `list` whose elements are objects of some class `Elem`. That is, we have:

```
ArrayList<Elem> list = ...
```

To traverse `list` the loop structure is:

```
for (Elem elt : list) {
    ...
}
```

Inside the loop, we can refer to the current element of the list as `elt`.

Traversing Lists in Java

In Java, we can also define recursive methods to traverse lists.

Java also has **patterns of recursion**, but they don't occur as higher-order functions.

Java's patterns of recursion occur as *language features*.

In particular, Java has a variety of **loop statements**.

In this course, we will concentrate on a single loop statement: the **“for each”** loop.

This loop corresponds exactly to a **simple list traversal**.

Example: Summing a List of Integers

Suppose we have a list of integers:

```
ArrayList<Integer> list
```

To sum them, we add each one in turn to a running total, like this:

```
Integer sum() {
    Integer total = 0;
    for (Integer x: list) {
        total = total + x;
    }
    return total;
}
```

Accumulating Parameters in Haskell

Loops like the one in `sum` above correspond closely to the approach of using **accumulating parameters** in Haskell definitions.

For example, a few weeks ago we looked at this version of a function to sum the elements of a list:

```
addUp []      total = total
addUp (x:xs) total = addUp xs (total+x)

sum xs = addUp xs 0
```

Other examples in `ListInt.java`

Accumulating Parameters (ctd)

Corresponding to the *parameter* `total` in the Haskell function, the Java version has a *variable* `total`.

In the Haskell version, we change the *total parameter* at each step to `(total+x)`. That is, we add the current element `x` to `total`.

In the Java version, we change the *total variable* at each step with the statement, `total = total+x`. That is, we add the current element `x` to `total`.

In the Haskell version, we initialise the parameter `total` to 0 in the “wrapper” function, `sum`.

In the Java version, we initialise the variable `total` to 0 with the assignment `total = 0`.

The next day . . .

Comparing Java Classes to Haskell Modules

In this lecture we will compare typical (good) Haskell and Java implementations of:

- Cartesian coordinate points
- Triangles (determined by their 3 vertices)
- Paths (i.e. sequences of points)

The aim is to emphasise their **commonalities** and their **differences**.

In Java, we will develop a **class** to represent points and the operations (methods) on points.

Since the class “is” the type, it will include **fields** to represent the data – i.e. the x and y components:

```
class Point {
    Double x;
    Double y;
    ..
}
```

Points

In Haskell, a good module structure will gather together the representation of points with the operations on points in to a **module**.

A natural representation of a point is as a **pair**:

```
module Point where
type Point = (Double, Double)
...
```

Constructors

In Haskell we can just write values and expressions of type **Point**. For example, $(0.0, 3.0)$.

In Java we must *explicitly construct* **objects** of class **Point**. The class has a **constructor**:

```
Point(Double xVal, Double yVal) {
    x = xVal;
    y = yVal;
}
```

which we use like this:

```
Point p = new Point(0.0, 3.0);
```

Accessing Fields

In Haskell, we can access the x and y components of values of type `Point` using **patterns**, e.g.:

```
distance (x1,y1) (x2,y2) = ...
```

In Java, we provide explicit **accessor methods**:

```
public Double xCoord() {  
    return x;  
}  
  
public Double yCoord() {  
    return y;  
}
```

The `translate` method is part of each `Point` object.

The `Translate` method *changes* the point.

The point is not passed as an argument to the method — the point being affected is *this point*.

The method does not return a result — it affects *this point*.

```
public void translate(Double xDist, Double yDist) {  
    x = x + xDist;  
    y = y + yDist;  
}
```

The method changes the x and y **fields** of this object.

Functions vs. Methods

In Haskell, we write **functions** to operate on points. For example, to compute a x,y translation of a point:

```
translatePt :: Double -> Double -> Point -> Point  
translatePt xDist yDist (x,y) = (x+xDist, y+yDist)
```

Notice that there is an **input** `Point` and an **output** `Point`, which is the result of the translation.

In Java we write **methods** to *manipulate* points.

`Point` objects may be *changed* by **mutator methods**.

Paths

A **path** is a sequence of points, so in Haskell a natural representation is as a **list** of points:

```
module Path where  
type Path = [Point]  
...
```

In Java, the data **field** in the `Path` class will be a list of points:

```
class Path {  
    ArrayList<Point> path;  
    ...
```

Constructing Paths

In Haskell we may simply write values of type `Path`, e.g.

```
[(0.0,0.0), (1.0,0.0), (0.0,1.0)]
```

In Java we must explicitly construct new `Path` objects:

```
Path() {  
    path = new ArrayList<Point>();  
}
```

which gives us an *empty path*.

Functions vs. Methods

In the Haskell `Point` module we wrote a function to translate a single point.

To translate a `Path` we want to apply the same translation to each point. The obvious approach is to use the `map` higher-order function:

```
translate :: Double -> Double -> Path -> Path  
translate xDist yDist path =  
    map (translatePt xDist yDist) path
```

The corresponding “pattern of recursion” in Java is the *for-each* loop.

We also need to provide a mutator method to build paths point by point:

```
public void extend(Point point) {  
    path.add(point);  
}
```

To build a path equivalent to the Haskell one

```
[(0.0,0.0), (1.0,0.0), (0.0,1.0)]:
```

```
Path myPath = new Path();  
myPath.extend(new Point(0.0,0.0));  
myPath.extend(new Point(1.0,0.0));  
myPath.extend(new Point(0.0,1.0));
```

(Whew!)

A loop beginning with:

```
for (Point point: path)
```

will traverse the `path` list. At each step, the current element of the list will be referenced by the `Point point` variable.

So applying the `translate` method to `point` inside the loop will have the same effect as `map` in Haskell: to apply the method to **every element of the list**:

```
void translate(Double xDist, Double yDist) {  
    for (Point point: path)  
        point.translate(xDist, yDist);  
}
```

Assignment 3

The representation of PPM images in Assignment 3 uses lists of lists of pixels, just as we did in Assignment 1. In Java it looks like this:

```
class PPM {  
    int width;  
    int height;  
    ArrayList<ArrayList<Pixel>> image;  
}
```

Most of the manipulations are list traversals, so the *for-each* loop is a good choice.

On the other hand, sometimes we want more control.

For example, in the *flip* manipulations we may want to go through the rows or pixels *in reverse order*.

In that case, the more general (and standard) version of the for-loop can be more appropriate.