Lecture 7: Accumulator methods, continued

Methods on trees, and accumulator-style methods on lists

Video Lessons

- Lesson 1 - Accumulators Part 1
- Lesson 2 - Accumulators Part 2

Overview

We continue with our ancestry trees example from Lecture 6, working through two subtle examples with accumulators, and then ending with a brief discussion of how accumulators can behave in potentially unexpected ways.

7.1 Finding the younger of two `IAT`s

In the section below, we're going to need to compare two `IAT`s and find the younger of them. This is similar to the `bornInOrBefore` method, except we're going to return an `IAT` instead of a boolean. We want a signature

```
// In IAT:
// To return the younger of this ancestor tree and the given ancestor tree
IAT youngerIAT(IAT other);
```

Let's work through the cases:

- If this `IAT` is `Unknown`, and the given `IAT` is `Unknown`, return `Unknown`.
- If this `IAT` is `Unknown`, and the given `IAT` is a `Person`, return the `Person`.
- If this `IAT` is a `Person`, and the given `IAT` is `Unknown`, return the `Person`.
- If this `IAT` is a `Person`, and the given `IAT` is `Person`, return the younger of the two `Person`s.

The first two cases are nice: if this `IAT` is `Unknown`, we just return the given `IAT`, regardless of what it is.

```
// In Unknown:
// To return the younger of this Unknown and the given ancestor tree
public IAT youngerIAT(IAT other) { return other; }
```

But the other two `Person` cases are harder, since the behavior depends on what the given `IAT` is.

```
// In Person:
// To return the younger of this Person and the given ancestor tree
public IAT youngerIAT(IAT other) {
    /* Template
     * Fields:
     * this.yob -- int
     * ... others as before
     * Methods:
     * this.youngerIAT(IAT) -- IAT
     * ... others as before
     * Methods on fields
     * ... others as before
     * Parameters:
     * other -- IAT
     * Methods on parameters
     * other.youngerIAT(IAT) -- IAT
     */
}
```

Again, inside this method we can't determine the age of the given `IAT`, since it might be `Unknown`. So we need a helper method that we invoke on the given `IAT` (like we invoked on `this.mom` above) where we pass along this `Person`'s age:

```
// In IAT:
// To return the younger of this ancestor tree and the given ancestor tree
IAT youngerIAT(IAT other);
// To return either this ancestor tree (if this ancestor tree is younger
// than the given yob) or the given ancestry tree
IAT youngerIATHelp(IAT other, int otherYob);
```

```
// In Unknown:
public IAT youngerIAT(IAT other) { return other; }
// To return either this Unknown (if this Unknown is younger than the
// given yob) or the given ancestry tree
IAT youngerIATHelp(IAT other, int otherYob) { return other; }
```

```
// In Person:
// To return the younger of this Person and the given ancestor tree
public IAT youngerIAT(IAT other) {
    /* Template
     * Fields:
     * this.yob -- int
     * ... others as before
     * Methods:
     * this.youngerIAT(IAT) -- IAT
     * this.youngerIATHelp(IAT, int) -- IAT
     * ... others as before
     * Methods on fields
     * ... others as before
     * Parameters
     * other -- IAT
```

```
      * Methods on parameters
      * other.youngerIAT(IAT) -- IAT
      * other.youngerIATHelp(IAT, int) -- IAT
      */
      return other.youngerIATHelp(this, this.yob);
  }
  // To return either this Person (if this Person is younger than the
  // given yob) or the given ancestry tree
  IAT youngerIATHelp(IAT other, int otherYob) {
      /* same template as above */
      if (this.yob > otherYob) {
          return this;
      }
      else {
          return other;
      }
  }
```

This implementation of `youngerIAT` for `Person` is quite strange at first sight.

> *Do Now!*
>
> Why should we pass both `this` and `this.yob`? And why should invoking a method on `other` be of any help here?

By invoking a method on `other`, we allow Java's dynamic dispatch to determine whether `other` is a `Unknown` or a `Person` — precisely the remaining question we had to figure out! We pass `this` along because it might be the desired result for the method. We pass `this.yob` because (like with `bornInOrBefore`) without it, `youngerIATHelp` would not be able to access the year of birth of the given `IAT`.

(Look at the purpose statement for `other.youngerIATHelp` very carefully, substituting "this" and "other" appropriately: If `other` is an `Unknown`, then `other.youngerIATHelp(this, this.yob)` will return *this `Person`*. On the other hand, if `other` is another `Person`, then `other.youngerIATHelp(this. this.yob)` will return *`other`* if `other` is younger than *this `Person`*, or else *this `Person`*.)

7.2 Finding the youngest grandparent

To determine the youngest grandparent of a given `IAT`, let's try a simpler method first: determine the youngest parent of a given `IAT`.

```
// In IAT:
// To compute the youngest parent of this ancestry tree
IAT youngestParent();
```

```
// In Unknown:
// To compute the youngest parent of this Unknown
public IAT youngestParent() { return new Unknown(); }
```

```
// In Person:
// To compute the youngest parent of this Person
public IAT youngestParent() {
    /* Template:
     * Fields:
     * this.mom -- IAT
     * this.dad -- IAT
     * ... others as before
     * Methods:
     * this.youngestParent() -- IAT
     * this.youngerIAT(IAT other) --- IAT
     * this.youngerIATHelp(IAT other, int otherYob) --- IAT
     * Methods of fields:
     * this.mom.youngestParent() -- IAT
     * this.mom.youngerIAT(IAT other) --- IAT
     * this.mom.youngerIATHelp(IAT other, int otherYob) --- IAT
     * this.dad.youngestParent() -- IAT
     * this.dad.youngerIAT(IAT other) --- IAT
     * this.dad.youngerIATHelp(IAT other, int otherYob) --- IAT
     */
}
```

This is fairly straightforward: we have `this.mom` and `this.dad`, and we have `youngerIAT` to return the younger of them:

```
// In Person:
// To compute the youngest parent of this Person
public IAT youngestParent() {
    return this.mom.youngerIAT(this.dad);
}
```

Now to implement `youngestGrandparent`, we can just return the `youngerIAT` of a `Person`'s parents' `youngestParents`:

```
// In IAT:
// To compute the youngest grandparent of this ancestry tree
IAT youngestGrandparent();
```

```
// In Unknown:
// To compute the youngest grandparent of this Unknown
public IAT youngestGrandparent() { return new Unknown(); }
```

```
// In Person:
// To compute the youngest grandparent of this Person
public IAT youngestGrandparent() {
    /* Template:
     * Fields:
     * this.mom -- IAT
     * this.dad -- IAT
     * ... others as before
     * Methods:
     * this.youngestParent() -- IAT
     * this.youngestGrandparent() -- IAT
     * this.youngerIAT(IAT other) --- IAT
```

```
     * this.youngerIATHelp(IAT other, int otherYob) --- IAT
     * Methods of fields:
     * this.mom.youngestParent() -- IAT
     * this.mom.youngestGrandparent() -- IAT
     * this.mom.youngerIAT(IAT other) --- IAT
     * this.mom.youngerIATHelp(IAT other, int otherYob) --- IAT
     * this.dad.youngestParent() -- IAT
     * this.dad.youngestGrandparent() -- IAT
     * this.dad.youngerIAT(IAT other) --- IAT
     * this.dad.youngerIATHelp(IAT other, int otherYob) --- IAT
     */
    return this.mom.youngestParent().youngerIAT(this.dad.youngestParent());
}
```

How can we generalize this to greatgrandparents and beyond? Notice that the implementations of `youngestParent` and `youngestGrandparent` are pretty similar: they both invoke `youngerIAT` on the youngest appropriate relative on `this.mom`'s side and the youngest appropriate relative on `this.dad`'s side.

We can simplify these methods with an accumulator, which keeps track of what number generation we're examining:

```
// this.andrew.youngestAncInGen(1) ==> this.andrew.youngestParent()
// this.andrew.youngestAncInGen(2) ==> this.andrew.youngestGrandparent()
// this.andrew.youngestAncInGen(3) ==> this.andrew.youngestGreatGrandparent()
// this.andrew.youngestAncInGen(0) ==> ?
```

The youngest `IAT` 1 generation away from `this.andrew` is `this.andrew`'s youngest parent. The youngest `IAT` 2 generations away from `this.andrew` is `this.andrew`'s youngest grandparent. The youngest `IAT` 0 generations away from `this.andrew` is...`this.andrew`!

Our code can then simplify to the following:

```
// In IAT:
// To compute the youngest ancestor in the given generation of this ancestry tree
IAT youngestAncInGen(int gen);
```

```
// In Unknown:
// To compute the youngest ancestor in the given generation of this Unknown
public IAT youngestAncInGen(int gen) {
    if (gen == 0) {
        return this;
    }
    else {
        return new Unknown();
    }
}
```

```
// In Person:
// To compute the youngest ancestor in the given generation of this Person
public IAT youngestAncInGen(int gen) {
    /* Template:
     * Fields:
     * this.mom -- IAT
```

```
     * this.dad -- IAT
     * ... others as before
     * Methods:
     * this.youngestAncInGen(int gen) -- IAT
     * this.youngerIAT(IAT other) --- IAT
     * this.youngerIATHelp(IAT other, int otherYob) --- IAT
     * Methods of fields:
     * this.mom.youngestAncInGen(int gen) -- IAT
     * this.mom.youngerIAT(IAT other) --- IAT
     * this.mom.youngerIATHelp(IAT other, int otherYob) --- IAT
     * this.dad.youngestAncInGen(int gen) -- IAT
     * this.dad.youngerIAT(IAT other) --- IAT
     * this.dad.youngerIATHelp(IAT other, int otherYob) --- IAT
     * Parameters:
     * gen -- int
     */
    if (gen == 0) {
        return this;
    }
    else {
        return this.mom.youngestAncInGen(gen - 1).youngerIAT(this.dad.youngestAncInGen(gen - 1));
    }
}
```

> *Do Now!*
>
> Complete the definition of `youngestGrandparent`, now that `youngestAncInGen` is defined.

The `gen` parameter is our accumulator, where this time it is "counting down" to decide at what depth to stop the recursion.

7.3 Potential hazards of accumulator-style methods

> **Exercise**
>
> Design a method `append` for lists of `String`s twice: first in direct style, and then again using an accumulator parameter.
>
> Do you notice any differences in the output? (Did you write enough tests?)