Lecture 6: Accumulator methods

Methods on trees, and accumulator-style methods on lists

Overview

This lecture practices several patterns of using delegation and helper methods. Each example gets more sophisticated than the previous, and keeping straight which objects are having their methods invoked, versus which objects are being passed as arguments, can be confusing. It helps to step through each method by hand, using the purpose statements to guide your understanding.

Let's go back to our ancestry trees example from Lecture 2, and design several methods on them. We'll enhance the `Person` class with some additional information – name, year of birth, gender — so that we have something to work with. Here are the skeletons of the relevant classes to get us started:

> Note that we're using a simplified representation of gender here—either male or female—and so we are using a `boolean` to represent it. *Given that,* we need to choose which boolean value to map to which gender, and our field name reflects whichever convention we pick. A fuller data definition would need a more inclusive datatype here, specifically an `enumeration`, but we haven't discussed that form of type definition in Java yet.

```
interface IAT {
}
class Unknown implements IAT {
   Unknown() { }
}
class Person implements IAT {
    String name;
    int yob;
    boolean isMale;
    IAT mom;
    IAT dad;
    Person(String name, int yob, boolean isMale, IAT mom, IAT dad) {
        this.name = name;
        this.yob = yob;
        this.isMale = isMale;
        this.mom = mom;
        this.dad = dad;
    }
}
```

Let's also assume that we have a list of strings to work with:

```
interface ILoString {

}
class ConsLoString implements ILoString {
    String first;
    ILoString rest;
    ConsLoString(String first, ILoString rest) {
        this.first = first;
        this.rest = rest;
    }
}
class MtLoString implements ILoString {
    MtLoString() { }
}
```

We want to answer the following questions about ancestry trees:

```
// To compute the number of known ancestors of this ancestor tree
// (excluding this ancestor tree itself)
int count();

// To compute how many ancestors of this ancestor tree (excluding this
// ancestor tree itself) are women older than 40 (in the current year)?
int femaleAncOver40();

// To compute whether this ancestor tree is well-formed: are all known
// people younger than their parents?
boolean wellFormed();

// To compute the names of all the known ancestors in this ancestor tree
// (including this ancestor tree itself)
ILoString ancNames();

// To compute this ancestor tree's youngest grandparent
IAT youngestGrandparent();
```

To do that, we'll need some example data and some expected test output:

```
class ExamplesIAT {
    IAT enid = new Person("Enid", 1904, false, new Unknown(), new Unknown());
    IAT edward = new Person("Edward", 1902, true, new Unknown(), new Unknown());
    IAT emma = new Person("Emma", 1906, false, new Unknown(), new Unknown());
    IAT eustace = new Person("Eustace", 1907, true, new Unknown(), new Unknown());

    IAT david = new Person("David", 1925, true, new Unknown(), this.edward);
    IAT daisy = new Person("Daisy", 1927, false, new Unknown(), new Unknown());
    IAT dana = new Person("Dana", 1933, false, new Unknown(), new Unknown());
    IAT darcy = new Person("Darcy", 1930, false, this.emma, this.eustace);
    IAT darren = new Person("Darren", 1935, true, this.enid, new Unknown());
    IAT dixon = new Person("Dixon", 1936, true, new Unknown(), new Unknown());

    IAT clyde = new Person("Clyde", 1955, true, this.daisy, this.david);
    IAT candace = new Person("Candace", 1960, false, this.dana, this.darren);
```

```
    IAT cameron = new Person("Cameron", 1959, true, new Unknown(), this.dixon);
    IAT claire = new Person("Claire", 1956, false, this.darcy, new Unknown());

    IAT bill = new Person("Bill", 1980, true, this.candace, this.clyde);
    IAT bree = new Person("Bree", 1981, false, this.claire, this.cameron);

    IAT andrew = new Person("Andrew", 2001, true, this.bree, this.bill);

    boolean testCount(Tester t) {
        return
            t.checkExpect(this.andrew.count(), 16) &&
            t.checkExpect(this.david.count(), 1) &&
            t.checkExpect(this.enid.count(), 0) &&
            t.checkExpect(new Unknown().count(), 0);
    }
    boolean testFemaleAncOver40(Tester t) {
        return
            t.checkExpect(this.andrew.femaleAncOver40(), 7) &&
            t.checkExpect(this.bree.femaleAncOver40(), 3) &&
            t.checkExpect(this.darcy.femaleAncOver40(), 1) &&
            t.checkExpect(this.enid.femaleAncOver40(), 0) &&
            t.checkExpect(new Unknown().femaleAncOver40(), 0);
    }
    boolean testWellFormed(Tester t) {
        return
            t.checkExpect(this.andrew.wellFormed(), true) &&
            t.checkExpect(new Unknown().wellFormed(), true) &&
            t.checkExpect(
                new Person("Zane", 2000, true, this.andrew, this.bree).wellFormed(),
                false);
    }
    boolean testAncNames(Tester t) {
        return
            t.checkExpect(this.david.ancNames(),
                new ConsLoString("David",
                    new ConsLoString("Edward", new MtLoString()))) &&
            t.checkExpect(this.eustace.ancNames(),
                new ConsLoString("Eustace", new MtLoString())) &&
            t.checkExpect(new Unknown().ancNames(), new MtLoString());
    }
    boolean testYoungestGrandparent(Tester t) {
        return
            t.checkExpect(this.emma.youngestGrandparent(), new Unknown()) &&
            t.checkExpect(this.david.youngestGrandparent(), new Unknown()) &&
            t.checkExpect(this.claire.youngestGrandparent(), this.eustace) &&
            t.checkExpect(this.bree.youngestGrandparent(), this.dixon) &&
            t.checkExpect(this.andrew.youngestGrandparent(), this.candace) &&
            t.checkExpect(new Unknown().youngestGrandparent(), new Unknown());
    }
}
```

> *Do Now!*
>
> Draw the ancestry tree for the example data above, and confirm that the test outputs are actually what we expect.

6.1 Counting

The `count` method is straightforward: every `Person` must count all their known ancestors. Every `Unknown` counts as zero. So we add the method `count` to the `IAT` interface, and then implement it in both classes. We start by specializing the purpose statement of the method to each class, and write down the template:

```
// In IAT:
// To compute the number of known ancestors of this ancestor tree (excluding this ancestor tree itself)
int count();
```

```
// In Unknown:
// To compute the number of known ancestors of this Unknown (excluding this Unknown itself)
public int count() { return 0; }
```

```
// In Person:
// To compute the number of known ancestors of this Person (excluding this Person itself)
public int count() {
    /* Template:
     * Fields:
     * this.name -- String
     * this.yob -- int
     * this.isMale -- boolean
     * this.mom -- IAT
     * this.dad -- IAT
     * Methods:
     * this.count() -- int
     * Methods of fields:
     * this.mom.count() -- int
     * this.dad.count() -- int
     */
}
```

Of all the items in our template, only the last two seem helpful for this method.

> *Do Now!*
>
> What exactly will invoking `this.mom.count()` produce? Specialize the purpose statement for `count` to `this.mom`.

Invoking `this.mom.count()` will produce the number of known ancestors of this `Person`'s `mom` (excluding this `Person`'s `mom` itself). Invoking `this.dad.count()` will be similar. Adding them together should produce the total number of known ancestors of this `Person`, as desired.

So our first try at implementing this method will be

```
// In Person:
public int count() {
    return this.mom.count() + this.dad.count();
}
```

> *Do Now!*
>
> Does this pass our tests? Why or why not?

Unfortunately, this always returns zero, because we never count any `Person` themselves; we only use their parents' counts, which eventually reach `Unknown` and just produce zero. (We do pass the test for `new Unknown().count()`, though!) So our second try should count this `Person` too:

```
// In Person:
public int count() {
    return 1 + this.mom.count() + this.dad.count();
}
```

> *Do Now!*
>
> Does this pass our tests? Why or why not?

We still fail our tests, but we're doing better: now our results are too large by one. We're counting the original `Person` that we invoked `count` upon. But if we can't add 1 for this `Person`, and we can't not add 1, what to do?

Since we need to treat the initial `IAT` that we invoke `count` upon specially from how we treat every subsequent `IAT`, we need a helper method. Our *original* method *will not* add 1 if it is invoked on a `Person`, but our *helper* method *will*. To achieve this we need to add the helper method to the interface also, and implement it on `Unknown` too:

```
// In IAT:
// To compute the number of known ancestors of this ancestor tree (excluding this ancestor tree itself)
int count();
// To compute the number of known ancestors of this ancestor tree (*including* this ancestor tree itself)
int countHelp();
```

```
// In Unknown:
// To compute the number of known ancestors of this Unknown (excluding this Unknown itself)
public int count() { return 0; }
// To compute the number of known ancestors of this Unknown (*including* this Unknown itself)
public int countHelp() { return 0; }
```

```
// In Person:
// To compute the number of known ancestors of this Person (excluding this Person itself)
public int count() {
    return this.mom.countHelp() + this.dad.countHelp();
}
```

```
// To compute the number of known ancestors of this Person (*including* this Person itself)
public int countHelp() {
    return 1 + this.mom.countHelp() + this.dad.countHelp();
}
```

> ***Do Now!***
>
> Now does this pass our tests? Why or why not?

6.2 Counting only some items

> ***Do Now!***
>
> Adapt the previous solution to `count`, to design the method `femaleAncOver40`.

Recall the purpose statement of `femaleAncOver40`: to compute how many ancestors of this ancestor tree (excluding this ancestor tree itself) are women older than 40 (in the current year). Since this has the same "treat the initial `IAT` differently from all ancestors" exception, we might well guess that we'll need a helper method here, too:

```
// In IAT:
// To compute how many ancestors of this ancestor tree (excluding this ancestor tree itself)
// are women older than 40 (in the current year).
int femaleAncOver40();
// To compute how many ancestors of this ancestor tree (*including* this ancestor tree itself)
// are women older than 40 (in the current year).
int femaleAncOver40Help();
```

```
// In Unknown:
// To compute how many ancestors of this Unknown (excluding this Unknown itself)
// are women older than 40 (in the current year).
public int femaleAncOver40() { return 0; }
// To compute how many ancestors of this Unknown (*including* this Unknown itself)
// are women older than 40 (in the current year).
public int femaleAncOver40Help() { return 0; }
```

```
// In Person:
// To compute how many ancestors of this Person (excluding this Person itself)
// are women older than 40 (in the current year).
public int femaleAncOver40() {
    /* Template:
     * Fields:
     * this.name -- String
     * this.yob -- int
     * this.isMale -- boolean
     * this.mom -- IAT
     * this.dad -- IAT
     * Methods:
     * this.count() -- int
     * this.countHelp() -- int
```

```
     * this.femaleAncOver40() -- int
     * this.femaleAncOver40Help() -- int
     * Methods of fields:
     * this.mom.count() -- int
     * this.mom.countHelp() -- int
     * this.mom.femaleAncOver40() -- int
     * this.mom.femaleAncOver40Help() -- int
     * this.dad.count() -- int
     * this.dad.countHelp() -- int
     * this.dad.femaleAncOver40() -- int
     * this.dad.femaleAncOver40Help() -- int
     */
}
// To compute how many ancestors of this Person (*including* this Person itself)
// are women older than 40 (in the current year).
public int femaleAncOver40Help() {
    /* same template as above */
}
```

Since the only difference between `count` and `femaleAncOver40` is that the latter is more selective in which `Person`s it counts, our code should be straightforward to adapt:

```
// In Person:
// To compute how many ancestors of this Person (excluding this Person itself)
// are women older than 40 (in the current year).
public int femaleAncOver40() {
    return this.mom.femaleAncOver40Help() + this.dad.femaleAncOver40Help();
}
// To compute how many ancestors of this Person (*including* this Person itself)
// are women older than 40 (in the current year).
public int femaleAncOver40Help() {
    if (2015 - this.yob > 40 && !this.isMale) {
        return 1 + this.mom.femaleAncOver40Help() + this.dad.femaleAncOver40Help();
    }
    else {
        return this.mom.femaleAncOver40Help() + this.dad.femaleAncOver40Help();
    }
}
```

> *Do Now!*
>
> Does this pass our tests? Why or why not?

> **Exercise**
>
> Design the method `numTotalGens`, which counts how many generations (including this `IAT`'s generation) are completely known. Design the method `numPartialGens`, which counts how many generations (including this `IAT`'s generation) are at least partially known. These methods should match the following behavior:
>
> ```
> boolean testNumGens(Tester t) {
>     return
> ```

```
        t.checkExpect(this.andrew.numTotalGens(), 3) &&
        t.checkExpect(this.andrew.numPartialGens(), 5) &&
        t.checkExpect(this.enid.numTotalGens(), 1) &&
        t.checkExpect(this.enid.numPartialGens(), 1) &&
        t.checkExpect(new Unknown().numTotalGens(), 0) &&
        t.checkExpect(new Unknown().numPartialGens(), 0);
}
```

6.3 Well-formedness

We define an ancestry tree to be well-formed if every `Person` in it is younger than its parents. We decide that all `Unknown`s are well-formed (they certainly aren't older than their parents!). Let's start our method templates, and see how far we can get:

```
// In IAT:
// To determine if this ancestry tree is well-formed
boolean wellFormed();
```

```
// In Unknown:
// To determine if this Unknown is well-formed
public boolean wellFormed() { return true; }
```

```
// In Person:
// To determine if this Person is well-formed
public boolean wellFormed() {
    /* Template:
     * Fields:
     * this.yob -- int
     * ... others as before
     * Methods:
     * ... others as before
     * this.wellFormed() -- boolean
     * Methods of fields:
     * ... others as before
     * this.mom.wellFormed() -- boolean
     * this.dad.wellFormed() -- boolean
     */
}
```

In the `Person` class, we need to compare this `Person`'s age (or yob) to their parents' ages...but we don't have the parents' ages available in our template! This isn't just stubbornness in the rules for templates; it's entirely possible that this `Person`'s parents might be `Unknown`, and therefore do not *have* ages.

One solution: age-related helper methods

Whenever we reach a situation where some object does not have enough information (between anything available in the template, or anything available via method parameters) to complete its task, it must delegate to a helper method — and sometimes, that helper method must be invoked

on a different object. Here, some `Person` cannot determine if it is younger than its parents. What to do?

- We could imagine creating a `getAge` method on `IAT`, so that we could ask any `IAT` what its age is...but there is no good answer to return for `Unknown`, so this approach won't work.
- We could try creating a `youngerThan(IAT)` method on `IAT`, so that we could ask any `IAT` whether it is younger than the given `IAT`. But this runs into the exact same problem, that we can't ask the given `IAT` what its age is.
- Asking person A the question "are you younger than person B?" is equivalent to asking person B "are you at least as old as person A?". So we could imagine creating a `atLeastAsOldAs(IAT)` method on `IAT`. This would let us define the method on `Unknown`s to return `true` (since `Unknown` ancestors are at least as old as known ones), but in the method on `Person`, again we cannot obtain the age of the given `IAT`.
- But! What if, instead of asking the question "are you at least as old as person A?", we ask the subtly different "are you at least as old as person A's age?" How does this help? Now, a `Person` can ask its parents whether they are older than its own age — after all, a `Person` knows its own age, and can pass that information along to where it is needed.

Let's try this last solution. Since we're recording ages as years of birth, we'll rephrase the question slightly, and add a method to `IAT`:

```
// In IAT:
// To determine if this ancestry tree is born in or before the given year
boolean bornInOrBefore(int yob);
```

```
// In Unknown:
// To determine if this Unknown is born in or before the given year
public boolean bornInOrBefore(int yob) { return true; }
```

```
// In Person:
// To determine if this Person is born in or before the given year
public boolean bornInOrBefore(int yob) {
    /* Template:
     * Fields:
     * this.yob -- int
     * ... others as before
     * Methods:
     * ... others as before
     * Methods of fields:
     * ... others as before
     */
     // Hooray -- we have all the information we need!
     return this.yob <= yob;
}
```

And now, we can complete `wellFormed` in `Person`:

```
// In Person:
// To determine if this Person is well-formed
public boolean wellFormed() {
```

```
    /* Template:
     * Fields:
     * this.yob -- int
     * ... others as before
     * Methods:
     * ... others as before
     * this.wellFormed() -- boolean
     * this.bornInOrBefore(int yob) -- boolean
     * Methods of fields:
     * ... others as before
     * this.mom.wellFormed() -- boolean
     * this.mom.bornInOrBefore(int yob) -- boolean
     * this.dad.wellFormed() -- boolean
     * this.dad.bornInOrBefore(int yob) -- boolean
     */
     return this.mom.bornInOrBefore(this.yob) &&
            this.dad.bornInOrBefore(this.yob) &&
            this.mom.wellFormed() &&
            this.dad.wellFormed();
}
```

Notice what happened: we invoked a method on `this.mom` and `this.dad`, passing in the field `this.yob`, because otherwise, `this.mom` would not have had access to that information (and neither would `this.dad`).

Another solution: accumulating the age information

In this particular case, the helper method `bornInOrBefore` seems a bit contrived; it's not likely to be of much use besides this particular problem. Also notice that the implementation of `wellFormed` is going to invoke `wellFormed` recursively on `this.mom` and `this.dad` — the same two objects on which we invoked `bornInOrBefore`. Perhaps we could combine both methods, performing the born-in-or-before check inside the well-formedness method and eliminating the helper?

One thing is certain: we can't eliminate that `yob` parameter that we worked so hard to obtain. So we'll have to build a helper method for `wellFormed` that takes in this extra parameter.

> *Do Now!*
>
> What does the `yob` parameter mean? When `bornInOrBefore(int yob)` was invoked in the code above, *whose* year of birth are we passing along?

From the perspective of a parent `Person`, the `yob` parameter in the `bornInOrBefore` method is their child's year of birth. So we tentatively produce the following purpose statement for our helper method:

```
// In IAT:
// To determine if this ancestry tree is well-formed
boolean wellFormed();
// To determine if this ancestry tree is older than the given year of birth,
// and its parents are well-formed
```

```
boolean wellFormedHelp(int childYob);
```

```
// In Unknown:
// To determine if this Unknown is well-formed
public boolean wellFormed() { return true; }
// To determine if this Unknown is older than the given year of birth,
// and its parents are well-formed
boolean wellFormedHelp(int childYob) { return true; }
```

```
// In Person:
// To determine if this Person is well-formed
public boolean wellFormed() {
    /* Template:
     * Fields:
     * this.yob -- int
     * ... others as before
     * Methods:
     * ... others as before
     * this.wellFormed() -- boolean
     * this.wellFormedHelp(int childYob) -- boolean
     * Methods of fields:
     * ... others as before
     * this.mom.wellFormed() -- boolean
     * this.mom.wellFormedHelp(int childYob) -- boolean
     * this.dad.wellFormed() -- boolean
     * this.dad.wellFormedHelp(int childYob) -- boolean
     */
}
// To determine if this Person is older than the given year of birth,
// and its parents are well-formed
boolean wellFormedHelp(int childYob) {
    /* same template as above */
}
```

In `Person`'s `wellFormedHelp` method, we certainly need to ask if this `Person` is older than the given year of birth, and we need to ask whether its parents are well formed. What year of birth should we pass up to the parents?

```
// In Person:
// To determine if this Person is older than the given year of birth,
// and its parents are well-formed
boolean wellFormedHelp(int childYob) {
    return this.yob <= childYob &&
           this.mom.wellFormedHelp(this.yob) &&
           this.dad.wellFormedHelp(this.yob);
}
```

Finally, we can fill in the definition of `wellFormed` itself: we don't need to check whether this `Person` is older than any given year of birth, but do need to check whether its parents are older than it is. (This is rather a similar "treat the first `IAT` differently than the rest" exception as in `count` and `femaleAncOver40` above.)

```
// In Person:
// To determine if this Person is well-formed: is it younger than its parents,
```

```
// and are its parents are well-formed
boolean wellFormed() {
    return this.mom.wellFormedHelp(this.yob) &&
           this.dad.wellFormedHelp(this.yob);
}
```

We call these extra parameters, that accumulate information as we progress through recursive calls, *accumulators*. This is a very powerful mechanism for improving the expressiveness of recursive methods, so Lecture 7 works through another example in this style.