Lecture 5: Methods for self-referential lists

Designing classes to represent lists. Methods on lists, including basic recursive methods, and sorting

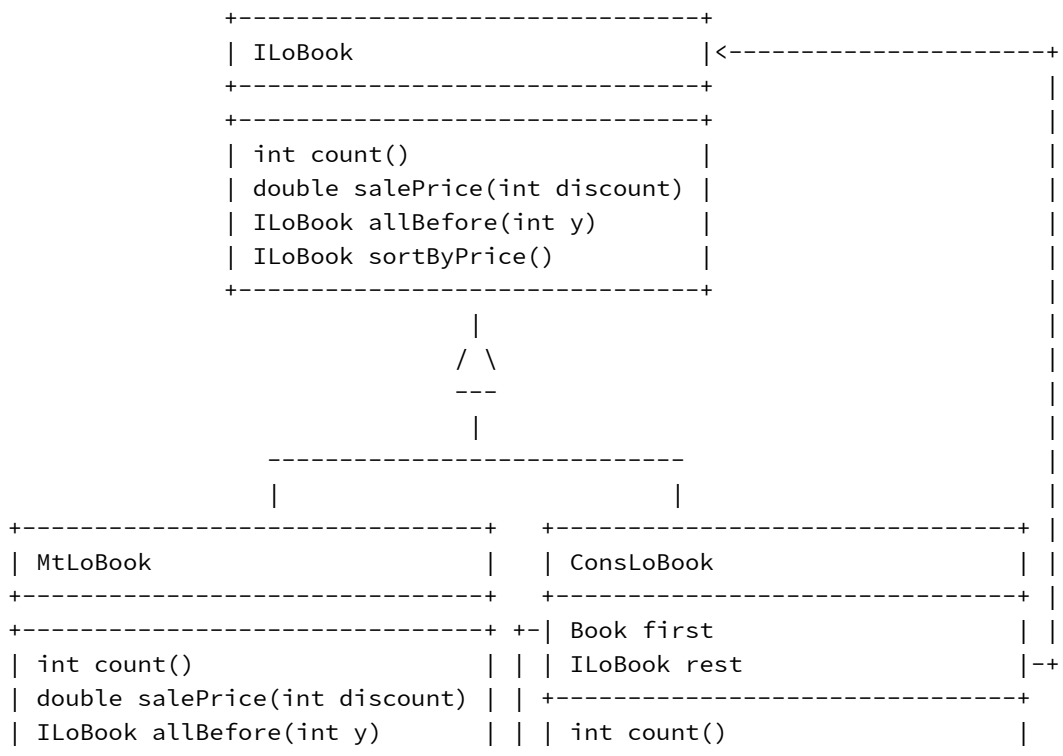
Related files:
BookLists.java

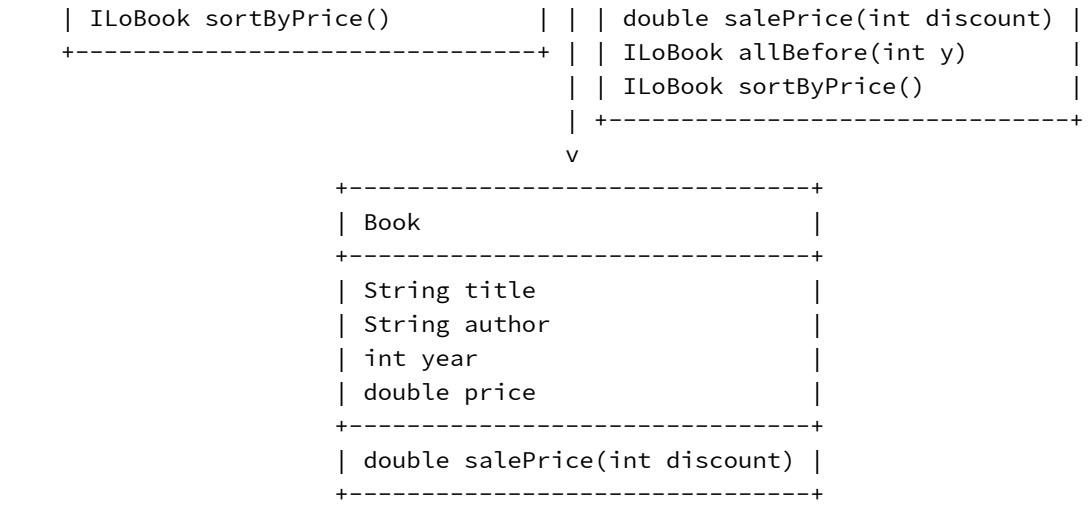
Lecture outline

- Representing lists
- Basic methods
- Sorting

5.1 Representing lists

The following class diagram defines a class hierarchy that represents a list of books in a bookstore:





Let's make some examples

```
//Books
Book htdp = new Book("HtDP", "MF", 2001, 60);
Book lpp = new Book("LPP", "STX", 1942, 25);
Book ll = new Book("LL", "FF", 1986, 10);

// lists of Books
ILoBook mtlist = new MtLoBook();
ILoBook lista = new ConsLoBook(this.lpp, this.mtlist);
ILoBook listb = new ConsLoBook(this.htdp, this.mtlist);
ILoBook listc = new ConsLoBook(this.lpp,
                new ConsLoBook(this.ll, this.listb));
ILoBook listd = new ConsLoBook(this.ll,
                  new ConsLoBook(this.lpp,
                    new ConsLoBook(this.htdp, this.mtlist)));
```

5.2 Basic list computations

Given this preceding class diagram, we would like to design methods to answer the following questions

- Count how many books we have in this list of books
- Compute the total sale price of all books in this list of books, at a given discount rate
- Given a date (year) and this list of books, produce a list of all books in this list that were published before the given year
- Produce a list of the same books as this list, but sorted by their price

Each of the four questions concerns a list of books, and so we start by designing the appropriate method headers and purpose statements in the interface `ILoBook`:

```
// In ILoBook
// -------
```

```
// count the books in this list
int count();

// produce a list of all books published before the given date
// from this list of books
ILoBook allBefore(int year);

// calculate the total sale price of all books in this list for a given discount
double salePrice(int discount);

// produce a list of all books in this list, sorted by their price
ILoBook sortByPrice();
```

We now have to define these methods in both the class `MtLoBook` and in the class `ConsLoBook`. (You may find it helpful to recall similar functions in DrRacket. Remember from last lecture the pattern given to us by virtue of Dynamic dispatch: we take each clause of the `cond` that checked for a particular variant and "move" the right-hand sides of those clauses into the methods defined in the corresponding class, then eliminate the `cond` altogether.)

The *design recipe* asks us to make examples. For clarity, we write them in an abbreviated manner, just showing the actual computation and the expected outcome:

```
// Examples for the class MtLoBook
// ------------------------------
mtlist.count() => 0
mtlist.salePrice(0) => 0
mtlist.allBefore() => mtlist
```

and our methods become:

```
// In MtLoBook:
// ---------

// count the books in this list
public int count() { return 0; }

// produce a list of all books published before the given date
// from this empty list of books
public ILoBook allBefore(int year) { return this; }

// calculate the total sale price of all books in this list for a given discount
public double salePrice(int discount) { return 0; }
```

Notice that the values produced by these methods are the base case values we have been in DrRacket for the empty lists. The `count` for an empty list is zero; the `salePrice` of no `Book`s is zero as well; and starting with an empty list there are no `Book`s at all, let alone any before a given year.

Note: We will return to the `sort` method later.

Of course, there will be more work to do in the `ConsLoBook` class. First, examples!

```
//  Examples
//  --------
lista.count() => 1
listc.count() => 3

lista.salePrice(0) => 25
listd.salePrice(0) => 95

lista.allBefore(2000) => lista
listb.allBefore(2000) => mtlist
listc.allBefore(2000) => new ConsLoBook(lpp, new ConsLoBook(ll,mtlist))
```

The *design recipe* asks us now to derive the template. A template serves as a starting point for **any** method inside `ConsLoBook`:

```
/*
TEMPLATE:
---------
Fields:
... this.first ...                          -- Book
... this.rest ...                           -- ILoBook
Methods:
... this.count() ...                        -- int
... this.salePrice(int discount) ...        -- double
... this.allBefore(int year) ...            -- ILoBook
Methods for Fields:
... this.rest.count() ...                   -- int
... this.rest.salePrice(int discount) ...   -- double
... this.rest.allBefore(int year) ...       -- ILoBook
*/
```

`count` and `salePrice`

In the template, `this.rest.count()` produces *the count of all books in the rest of this list* — and so the method body in the class `ConsLoBook` becomes:

```
// count the books in this list
public int count() { return 1 + this.rest.count(); }
```

In the template, `this.rest.salePrice(discount)` produces *the total sale price of all books in the rest of this list for the given discount* —and so the method body in the class `ConsLoBook` just adds to this value the price of the first book in the list:

```
// calculate the total sale price of all books in this list for the given discout
public double salePrice(int discount) {
  return this.first.salePrice(discount) + this.rest.salePrice(discount);
}
```

> ***Do Now!***

> Did you notice how similar this method body is to the one above for `count`? In Fundies 1, we had a terser way of expressing this sort of computation. What kind of operation on lists are we computing here? We will see in Lecture 13: Abstracting over behavior how to improve this code.

---

allBefore

In the template, `this.rest.allBefore(year)` produces *a list of all books in the rest of this list published before the given date*. The only work that remains is to decide whether the *first* book of this list belongs in the output list, and either add it to the result or not. If only we could determine whether that first `Book` was published before the given year! (Look carefully at the template: we cannot access `this.first.year`, because we do not have access to fields of fields.) So we add a method to our wish list, and we will delegate the job of deciding this question to the `Book` class itself. The method body in the class `ConsLoBook` becomes:

```
// produce a list of all books published before the given date
// from this non-empty list of books
ILoBook allBefore(int year) {
  if (this.first.publishedBefore(year)) {
    return new ConsLoBook(this.first, this.rest.allBefore(year));
  }
  else {
    return this.rest.allBefore(year);
  }
}
```

(This method introduces a new piece of syntax: *if statements*. An if-statement always follows the same basic template:

```
if (some condition) {
  //...statements to execute if condition was true...
}
else {
  //...statements to execute if condition was false...
}
```

where only one of the branches of the `if` executes its statements. Note that unlike DrRacket, an `if` in Java is *not* an expression, and does not produce a value. In the code for `allBefore` above, the `if` statement itself does not return a value; the `return` statements inside it do.)

We're not quite done; we have a method remaining on our wish list, so we must add to the class `Book` the method

```
// was this book published before the given year?
boolean publishedBefore(int year) {
  return this.year < year;
}
```

> **Exercise**
>
> Flesh out the rest of the design of this method, adding examples and tests.

Of course, for all of these methods, we end the design process by making sure all tests run. The actual test methods will be:

```
// tests for the method count
boolean testCount(Tester t) {
  return
  t.checkExpect(this.mtlist.count(), 0) &&
  t.checkExpect(this.lista.count(), 1) &&
  t.checkExpect(this.listd.count(), 3);
}

// tests for the method salePrice
boolean testSalePrice(Tester t) {
  return
  // no discount -- full price
  t.checkInexact(this.mtlist.salePrice(0), 0.0, 0.001) &&
  t.checkInexact(this.lista.salePrice(0), 10.0, 0.001) &&
  t.checkInexact(this.listc.salePrice(0), 95.0, 0.001) &&
  t.checkInexact(this.listd.salePrice(0), 95.0, 0.001) &&
  // 50% off sale -- half price
  t.checkInexact(this.mtlist.salePrice(50), 0.0, 0.001) &&
  t.checkInexact(this.lista.salePrice(50), 5.0, 0.001) &&
  t.checkInexact(this.listc.salePrice(50), 47.5, 0.001) &&
  t.checkInexact(this.listd.salePrice(50), 47.5, 0.001);
}

// tests for the method allBefore
boolean testAllBefore(Tester t) {
  return
  t.checkExpect(this.mtlist.allBefore(2001), this.mtlist) &&
  t.checkExpect(this.lista.allBefore(2001), this.lista) &&
  t.checkExpect(this.listb.allBefore(2001), this.mtlist) &&
  t.checkExpect(this.listc.allBefore(2001),
     new ConsLoBook(this.lpp, new ConsLoBook(this.ll, this.mtlist))) &&
  t.checkExpect(this.listd.allBefore(2001),
     new ConsLoBook(this.ll, new ConsLoBook(this.lpp, this.mtlist)));
}
```

5.3 Sorting

The last method to design was defined in the interface `ILoBook` as:

```
// produce a list of all books in this list, sorted by their price
ILoBook sortByPrice();
```

An empty list is sorted already, so in the class `MtLoBook` the method becomes:

```
// produce a list of all books in this list, sorted by their price
public ILoBook sortByPrice() {
  return this;
}
```

We do not need to create a `new` empty list, `this` one works perfectly well.

We need examples for the more complex cases. We recall our sample data:

```
//Books
Book htdp = new Book("HtDP", "MF", 2001, 60);
Book lpp = new Book("LPP", "STX", 1942, 25);
Book ll = new Book("LL", "FF", 1986, 10);

// lists of Books
ILoBook mtlist = new MtLoBook();
ILoBook lista = new ConsLoBook(this.lpp, this.mtlist);
ILoBook listb = new ConsLoBook(this.htdp, this.mtlist);
ILoBook listc = new ConsLoBook(this.lpp,
                new ConsLoBook(this.ll, this.listb));
ILoBook listd = new ConsLoBook(this.ll,
                  new ConsLoBook(this.lpp,
                    new ConsLoBook(this.htdp, this.mtlist)));
ILoBook listdUnsorted =
  new ConsLoBook(this.lpp,
    new ConsLoBook(this.htdp,
      new ConsLoBook(this.ll, this.mtlist)));
```

and our tests will be:

```
// test the method sort for the lists of books
boolean testSort(Tester t) {
  return
  t.checkExpect(this.listc.sortByPrice(), this.listd) &&
  t.checkExpect(this.listdUnsorted.sortByPrice(), this.listd);
}
```

Next we look at the template that is relevant for this question:

```
/*
TEMPLATE:
---------
Fields:
... this.first ...                          -- Book
... this.rest ...                           -- ILoBook
Methods:
... this.sortByPrice() ...                  -- ILoBook
Methods for Fields:
... this.rest.sortByPrice() ...             -- ILoBook
*/
```

Reading the purpose statement for `sortByPrice` carefully, we see that (like `allBefore` above) `this.rest.sortByPrice()` does almost all the work for us — it produces a sorted rest of this list. What makes this method more challenging than `allBefore` is that we aren't simply prepending to the beginning of that resulting list; we need to *insert* the first element of the list into its appropriate place in the sorted rest of the list. This sounds like a job for a helper method, so we add it to our wish list and move on.

> ***Do Now!***
>
> When we do get to it, where should this helper method be defined?

Implementing `sortByPrice` for `ConsLoBook` is now straightforward: we just translate the English sentence above into Java.

```
// In ConsLoBook
// produces a list of the books in this non-empty list, sorted by price
public ILoBook sortByPrice() {
  return this.rest.sortByPrice()  // sort the rest of the list...
             .insert(this.first); // and insert the first book into that result
}
```

Now we need to finish off the items on our wish list. We need `insert` to produce a list whose contents are the same as the contents of this (already sorted!) list, but with the given `Book` inserted into its proper place. To do this, we'll certainly need to compare whether one book is cheaper than another, so we'll add that to our wish list and move on.

> ***Do Now!***
>
> Implement `insert` for `ConsLoBook`. Pay careful attention to the use of the template to guide your recursive calls.

If we define `insert` as a method in `ConsLoBook`...

```
// in ConsLoBook
// insert the given book into this list of books
// already sorted by price
public ILoBook insert(Book b) {
  if (this.first.cheaperThan(b)) {
    return new ConsLoBook(this.first, this.rest.insert(b));
  }
  else {
    return new ConsLoBook(b, this);
  }
}
```

...Java complains.

> ***Do Now!***

> Why? What did we forget? (Hint: if you try this code in Eclipse, where does it indicate there are errors?)

Yes, we haven't implemented `cheaperThan` yet. Let's fix that right now:

```
// in Book
// is the price of this book cheaper than the price of the given book?
boolean cheaperThan(Book that) {
  return this.price < that.price;
}
```

But still we have a problem. We've defined `insert` on `ConsLoBook`, but in the third line, we write `this.rest.insert(b)` — and we do not know anything about `this.rest` except that it's an `ILoBook`, and `ILoBook` says nothing about an `insert` method!

Ok let's add the method to our interface:

```
// in ILoBook
// insert the given book into this list of books
// already sorted by price
ILoBook insert(Book b);
```

And now we have fixed our problem here, only to create a new problem elsewhere.

> *Do Now!*
>
> *Now* what did we miss?

This is another, subtle example of the benefits of writing down our types explicitly. In DrRacket, if we tried to define a function over a union type, and *forgot* a case, the only way we'd find out is if a test caught the lapse. Here, Java can immedately warn us that we've forgotten something

Now that we've promised that all `ILoBook`s must implement `insert`, we need to implement it on `MtLoBook` too. How can we insert a `Book` into its proper place in an empty list? By building a list with only the given book in it:

```
// in MtLoBook
// insert the given book into this empty list of books
// already sorted by price
public ILoBook insert(Book b) {
  return new ConsLoBook(b, this);
}
```

And now we are finally done!

This sorting order is called "lexicographic", and is the generalization of sorting alphabetically to account for digits, punctuation, other alphabets, and all the other characters allowed in strings.

Exercise

Suppose we wanted to sort the books by title, instead of by price. We cannot use the `<` operator to compare `String`s. Instead, `String`s have a method `compareTo(String)` that returns:

- A negative integer if this `String` is lexicographically before the given `String`
- 0 if the strings are lexicographically equal
- A positive integer if this `String` is lexicographically after the given `String`

Use this method to define a method `titleBefore` on `Book`s, analogous to `cheaperThan`, and revise `sort` and/or `insert` to use it.