Designing Recursive Methods with Accumulators

Our goal is to practice converting the structural recursion into recursion that uses accumulators to keep track of the previous knowledge.

The code in the lecture from January 24th implements the following methods for classes that represent a list of songs:

// Count the number of songs in ∗this∗ LoS
int count();

// Find the total size of all the songs in ∗this∗ LoS
int totalSize();

// Is a song by the given artist in ∗this∗ LoS
boolean contains(*String artist*);

// Create a list of all songs by the given artist in ∗this∗ LoS
LoS allBy(*String artist*);

We now convert our methods to use the accumulator style. As we did in Scheme, we will design a helper method that takes one more argument, the accumulator. The accumulator at any given point in the computation will hold the "answer" to our question thus far.

The first question to ask is:

Step 1: What is the type of the expected result?

This determines not only the return type for the method, but also the type of the value that will be accumulated. If the type is ¡code¿*AccType*, we add an argument *AccType acc* to the method signature and add *Acc* to its name.

That means we add the following methods to the interface:

// Count the number of songs in ∗this∗ LoS
int countAcc(*int acc*);

// Find the total size of all the songs in ∗this∗ LoS
int totalSizeAcc(*int acc*);

// Is a song by the given artist in ∗this∗ LoS
boolean containsAcc(*String artist*, *boolean acc*);

// Create a list of all songs by the given artist in ∗this∗ LoS
LoS allByAcc(*String artist*, *LoS acc*);

The next step is to change the purpose statements to explain the meaning of the accumulated value.

Step 2: Change the purpose statements for all new methods.

For example, we say:

// add to the acc the number of songs in ∗this∗ LoS
int countAcc(*int acc*);

Next we need to figure out what are the values each method produces when it is invoked withe the empty list. Think of where can you find out. We call this the **base value**.

Step 3: Make a list of the base values for all new methods.

Of course, the base value for the *count* method is `0` .

We now deal with the following problem. The original question did not mention the accumulator. That means that the method that uses the accumulator is only our helper method. The original method needs to invoke this helper so that it would produce the original result.

There are two steps that make this possible.

Step 4: Invoke the method that uses the accumulator using the base value as the initial accumulator value.

For example we get:

```
// Count the number of songs in ∗this∗ LoS
int count(){
   return this.countAcc(0);
}
```

As the things stand now, we need to add this method body to the method definition in both the empty class (*MtLoS*) and the class that represents a nonempty list (*ConsLoS*). In the next section we will learn how to do this only once.

The next step deals with the class that represents the empty list.

Step 5: Implement the method body in the class MtLoS.

The purpose statement makes it clear that the method just produces the value of the accumulator. This makes sense. If the original method is invoked by an instance of the empty class, it produces the base value, as expected.

Implement all method bodies for the class *MtLoS*.

Now comes the hardest part of the whole process.

Step 6: Design the update method.

Let us examine what happens inside of the method body in the class *ConsLoS*. We list all method definitions here:

```
// Count the number of songs in *this* non-empty LoS
int count(){ return 1 + this.rest.count();
}

// Find the total size of all the songs in *this* non-empty LoS
int totalSize(){  return this.first.size + this.rest.totalSize();
}

// Tell if there is a song by the given artist in *this* non-empty LoS
boolean contains(String artist){
   return this.first.artist.equals(artist) ||
            this.rest.contains(artist);
}

// Create a list of all songs by the given artist in *this* non-empty LoS
LoS allBy(String artist){
   if(this.first.artist.equals(artist))
     return new ConsLoS(this.first, this.rest.allBy(artist));
   else
      return this.rest.allBy(artist);
}
```

In each case, there is some computation that involves *this.first* and the recursive invocation of the original method by the list *this.rest*, though in the *count* method it only contributes 1 to the count.

The *update* method has the following common structure:

// update the value of the accumulator using this.first value
AccType updateMethodName(*Song first*, *AccType acc*){ ... }

For the method *count* the *updateCount* method is defined as:

// update the value of the count by adding the first item to the count
int updateCount(*Song first*, *int acc*){ *return* 1 + *acc*;
}

Design the remaining *update* methods. And, yes, *do follow the design recipe*.

We are just about done.

Step 7: Complete the body of the method using the update method.

The method that uses the accumulator is the same for all cases. The instance of the *rest* of the list invokes the method self-referentially, using the updated value of the accumulator.

For the *count* method this will be:

// add to the acc the number of songs in ∗this∗ LoS
int countAcc(*int acc*){
  return this.rest.countAcc(*this.updateCount*(*this.first*, *acc*));
}

While this may look to be very complicated for the simple *count* method, it is much more useful and cleaner for the remaining methods.

Finish designing the bodies of the remaining methods. When done, run the original tests for the main methods as well as all of your other test cases.

Quiz

You have 10 minutes.

Abstracting over Data Definitions

A bank customer can have three different accounts: a checking account, a savings account, and a line of credit account.

- The customer can withdraw from a checking account any amount that will still leave the minimum balance in the account. The customer can withdraw all money from a savings account. The balance

of the credit line represents the amount that the customer already borrowed against the credit line. The customer can withdraw any amount that does not make the balance exceed the credit limit.

- The customer can deposit money to the account in any amount. If the customer deposits more to the credit line than the current balance, the balance will become negative — indicating overpayment.

4.1 Review of Designing Methods for Unions of Classes.

The code in *lab4-banking.bjava* defines the classes that represent this information.

1. Make examples of data for these classes, then make sure you understand the rules for withdrawals.

   Now design the methods that will manage the banking records:

2. Design the method *canWithdraw* that determines whether the customer can withdraw some desired amount.

3. Design the method *makeDeposit* that allows the customer to deposit a given amount of money into the account.

4. Design the method *maxWithdrawal* that computes the maximum that the customer can withdraw from an account.

5. Design the method *moreAvailable* that produces the account that has more money available for withdrawal.

4.2 Abstracting over Data Definitions: Lifting Fields

Save your work and open it again with the file type 'ijava'. Change the language level to Intermediate ProfessorJ.

Look at the code and identify all places where the code repeats — the opportunity for abstraction.

Lift the common fields to an abstract class *ABanking*. Make sure you include a constructor in the abstract class, and change the constructors in the derived classes accordingly. Run the program and make sure all test cases work as before.

4.3 Abstracting over Data Definitions: Lifting Methods

For each method that is defined in all three classes decide to which category it belongs:

1. The method bodies in the different classes are all different, and so the method has to be declared as *abstract* in the *abstract* class.
2. The method bodies are the same in all classes and it can be implemented completely in the *abstract* class.
3. The methods look very similar, but each produces a different variant of the union — therefore it cannot be lifted to the *super* class.
4. The method bodies are the same for two of the classes, but are different in one class — therefore we can define the common body in the *abstract* class and override it in only one derived class.

Now, lift the methods that can be lifted and run all tests again.

Part 2: Designing the Pong Game

A Game of Pong

Rules:

A ball starts at a random height on the left and falls from the left side diagonally down. At the bottom is a paddle that can move left and right controlled by the arrow keys.When the ball hits the bottom, but misses the paddle, it disappears from the game. When the ball hits the paddle, it bounces back and continues diagonally up to the right. When the ball exits the playing field, a new ball comes into play.

Classes needed:

Ball - a Posn and the direction in which the ball moves (up or down)
Paddle - a Posn

PongWorld - contains one Ball and one Paddle, has a fixed width and height

The code in the file *pong-game-skeleton.ijava* defines the classes that represent the ball, the paddle, and the world. (For now, we ignore the world).

The class *PongWorld* extends the class *World* in the teachpack. Therefore, we need to use *ProfessorJ Intermediate Language*.

4.4 Designing methods in the class *Ball*

1. Design the method *draw* that displays the ball on a canvas. The following code (that can be written within the *Examples* class shows how you can draw one circle:

```
import draw.*;
import colors.*;
import geometry.*;

class Examples{
  Examples() {}

  Canvas c = new Canvas(200, 200);

  boolean makeDrawing =
    this.c.show() &&
    this.c.drawDisk(new Posn(100, 150), 50, new Red());
}
```

   The three *import* statements on the top indicate that we are using the code programmed by someone else and available in the libraries named *draw*, *colors*, and *geometry*. Open the *Help Desk* and look under the *Teachpacks* for the teachpacks for *How to Design Classes* to find out more about the drawing and the *Canvas*.

2. Design the method *moveBall* that moves the ball five pixels in its direction.

3. Design the method *bounce* that produces a ball after it bounced up to the right, and with its direction set to move up to the right.

4. Design the method *hitBottom* that determines whether the ball hit the bottom of the canvas of the given height.

5. Design the method *outOfBounds* that determines whether the ball is out of bounds of the canvas of the given width and height.

4.5 Designing methods in the class *Paddle*

1. Design the method *draw* that displays the paddle on a given canvas of the given height.

2. Design the method *movePaddle* that consumes a *String* and moves the paddle either left or right depending on the *String* it receives as argument. For now, ignore the requirement that the paddle stays within the bounds of the canvas. (You may add it later, once the program is working.)

3. Design the method *hitBall* that determines whether the paddle hit the given ball. For simplicity, just make sure that the distance between the center of the ball and the center of the top of the paddle is less than or equal to the radius of the ball. You may need to delegate the work to the class *Ball*.

4.6 Designing methods in the class *PongWorld*

1. Design the methods *draw* and *erase* that show the background, the ball, and the paddle. Make the background black.

2. Finally, add some interactions to your program, by letting the paddle move in response to the key events. Design the method *onKeyEvent* that consumes a *String* and moves the paddle left, or right by 5 pixels every time the user hits one of the corresponding arrow keys.

   Use the code in the program *WorldDemo.ijava* to figure out how to respond to the key events and to see how to run the program.

3. Design the method *onTick* as follows:

   - If the ball is out of bounds, replace the ball with a new ball. Initially, start the new ball in the top left corner. When everything else is working, make the ball start on the left edge at a random height between the top and the middle.
   - If the ball hit the paddle, replace the ball with a new one going in the opposite direction - and moved ahead.
     Hint: Recall the method bounce in the class Ball.

- Otherwise, just move the ball in its direction.
- Currently the world never ends. When all is working, think of what may be the appropriate end of the game (count the number of balls that were put into play, the number of balls that hit the paddle, or the number of elapsed ticks. Then modify the appropriate methods so that the world eventually ends.

Save all your work — the next lab may build on the work you have done here!