Computer Science 252
Problem Solving with Java
The College of Saint Rose
Spring 2014

Topic Notes: Recursive Structures and Methods

Recursion

Suppose you are asked to construct a new graphical object called "nested squares". Such an object would consist of a square enclosing a series of smaller squares. We can do this:

See Example: NestedSquaresLoop

But there's another way we can think about this problem. Rather than having a loop that draws our all of our nested squares, we can break this down into two steps:

1. Draw the outermost square.
2. Draw the rest of the squares, which is itself a set of nested squares.

This is a *recursive* way to approach this task. And we can do just this in our program:

See Example: NestedSquaresRec

At first, this might look like a ridiculous thing to do. How can we construct a `NestedSquaresRec` object by constructing one square, then another `NestedSquaresRec` object? Isn't that like using the word you're defining in a definition? How do we stop?

The key here is that we only draw another square as long as its size is positive. Since each recursive call to the `NestedSquaresRec` is passing a smaller value for the `size` parameter, we'll eventually get to a case where `size` is not greater than 0, in which case our constructor will do nothing at all.

Recursive Data Structures

Now let's make things a bit more interesting. Suppose that we want to be able to move the nested squares in the same way we do a filled or framed rectangle. That is easy to do with this recursive version as long as we have names for the objects we create. We can then use these names in our usual methods.

See Example: NestedSquaresDrag

The only change we have made to the constructor is to give names to the square (now called `outline`) and the rest of the nested squares, `rest`. The reason for this is so that we can refer to them in the other methods as needed.

Notice that the type of rest is `NestedSquares`. Java has no problem with an instance variable whose type is the same as the entire class. When a class has an instance variable with the same type as the class, we say that it represents a *recursive data structure*.

The `contains` method is straightforward since we only need to check if the `outline` contains the point.

The `move` method requires a little more effort, but it is still very straightforward when you think about it. We simply move `outline` by the requested offsets, and then move `rest` by those same offsets. Other methods like `setColor`, `moveTo`, and `removeFromCanvas` would be written similarly.

The basic idea is this: since we have a recursive data structure, methods manipulating it will also be recursive. In this case, we have taken care of the non-recursive part (`outline`) and then send a recursive message to the `rest`.

How could you do this if you didn't want to do it recursively? If you were to use a while loop you would need to save the names of all of the squares! We'll see soon that this can be a perfectly reasonable solution, using arrays or `ArrayList`s, but it's certainly more complex than our recursive solution.

As an additional example, we will complete the development of a generalized snowman, together in class.

See Example: BigSnowman

Now, consider this extension of the "nested squares" idea.

See Example: NestedSquaresSizes

Here, we enhance the functionality of the `WindowController` class to add some Swing components, and the `NestedSquares` recursive object with a few new recursive methods.

First, in the `NestedSquaresSizes` class, the Swing components include two that are familiar: `JButton`s to allow us to set the color of and retrieve some information about the most recently created nested squares, and a `JSlider` that allows us to control the size of the next nested squares to be created.

The new items in use here are `JLabel`s, which are probably the simplest of the Swing components. We create them and add them to containers (one to the `BorderLayout.NORTH`, two into a `JPanel` that will be added to the `BorderLayout.SOUTH`). Two are just included to give some indication of the lower and upper limits of the slider, and the other will later be used to display some information when the "Get Info" button is pressed.

The main functionality is similar to previous examples: if the mouse is pressed anywhere other than inside the most-recently created `NestedSquares`, a new one is created, with a size as determined by the value of the slider. If pressed in that most recently-created one, we start a standard dragging operation.

The buttons both operate on the most-recently created object. The "Random Color" button sets the color of the newest object to a randomly-chosed new color. The "Get Info" button asks the newest

object for two pieces of information: the number of squares it consists of, and the total perimeter of all of the squares within the newest object. Those bits of information are displayed back to the user in the `JLabel`, updated with the `setText` method.

The more interesting parts of the example are the new recursive methods of `NestedSquares`. The `setColor` method is very similar to the `move` method we saw before. To set the color of the whole object, we set the color of the outermost square (`outline`) then if there are more squares referred to by `rest`, we set the color of the rest recursively.

The other two methods are a bit different. Rather than being mutator methods like the ones above, these are accessor methods: they compute and return something.

First, `squareCount`. This method returns the total number of squares (`FramedRect`s) that make up the object. In many cases, an accessor method simply returns the value of an instance variable, or maybe some simple arithmetic expression based on a few instance variables. But since this structure was created and is stored in memory recursively, the answer to this seemingly simple question is not readily available. So we need a recursive approach to this method. Our approach is just like before, when we create or modify the object by taking care of the outermost square, then the rest. Here, we count the number of squares that make up the outermost square (1, obviously), and then figure out how many squares are used within the rest (which is exactly what we are writing a method to compute, conveniently enough).

The `totalPerimeter` method is similar, except there is a bit more work to do to compute the perimeter of the outermost square.

In both of these cases, we proceed similarly: we figure out the contribution to the total of the outermost square (1 for `squareCount`, the perimeter of the outermost square for `totalPerimeter`). If there is no "rest", we simply return that value. If there is, we recursively compute the answer for the rest, then add that to the contribution of the outermost.

Recursive Functions

The last 2 methods lead us to the more general notion of a recursive function. We saw several examples tied to `NestedSquares`, but they occur in many domains. Let's look at a program to raise numbers to powers:

See Example: Powers

This program is a Java application, more like the programs you saw prior to this course. Specifically, it does not bring up a graphics window at all, and its execution starts in a `main` method instead of a `begin` method.

Putting all of that aside, we can see that the basic idea here is to read in a couple of integers, a base and an exponent, and then raise the base to that power. There are three methods here, all of which compute the same thing but in different ways.

The first, `loopPower` simply contains a `for` loop to multiply `base` by itself `exponent` times. Correct, but not especially interesting.

We will focus on the others, first `recPower`. How does this one work? Clearly, if we evaluate `recPower(3,0)`, the condition `exponent==0` is true, so the function should return 1. Suppose instead we evaluate `recPower(3,1)`. According to the function definition and the fact that `1!=0`, we get that `recPower(3,1) = 3*recPower(3,0)`, and we know the value of `recPower(3,0)` is 1. Thus the final answer is `3*1` or `3`. The key is that we are using the facts that $b^0 = 1$ and $b^{e+1} = b*b^e$ to calculate powers. Because we are calculating complex powers using simpler powers we eventually get to our base case.

A handy way of thinking about recursive programs is to think about having someone else handle the recursive call. I.e., if I want to calculate `recPower(3,5)`, I ask someone else to calculate `recPower(3,4)` and then, when they give me the answer, `81`, multiply that answer by 3 to get the final answer of `243`.

Using a simple modification of the above recursive function we can get a very efficient algorithm for calculating powers, shown in `fastRecPower`. In particular, if we use either of the first two functions, it will take 1024 multiplications to calculate 3^{1024}. Using a slightly cleverer algorithm we can cut this down to only 11 multiplications!

In each of the first two functions, the number of multiplications necessary is equal to the value of the exponent. That is not the case here.

```
fastRecPower(3,16) = fastRecPower(9,8)                    // mult
    = fastRecPower(81,4)                 // mult
    = fastRecPower(6561,2)               // mult
    = fastRecPower(43046721,1)           // mult
    = 43046721 * fastRecPower(43046721,0)
    = 43046721 * 1                     // mult
    = 43046721
```

Thus it only took 5 multiplications (and 4 divisions by 2) using `fastRecPower`, whereas it would have taken 16 multiplications the other way (and divisions by two can be done very efficiently in binary).

In general it takes somewhere between $log_2(\texttt{exponent})$ and $2 * log_2(\texttt{exponent})$ multiplications to compute a power this way. While this doesn't make a difference for small values of exponent, it does make a difference when exponent is large. For example, computing `fastRecPower(3,1024)` would only take 11 multiplications, while computing it the other way would take 1024 multiplications.

Why does this algorithm work? It works because it is based on the following simple rules of exponents:

- $base^0 = 1$
- $base^{exp+1} = base * base^{exp}$
- $base^{2*exp} = (base^2)^{exp}$

The key is that by rearranging the order of doing things in a clever way, we can cut down the amount of work considerably! (Again it is possible to write the above algorithm with a while loop, but the above formulation is arguably easier to understand!)

We can both write and understand recursive programs as follows:

1. Write the base case. Convince yourself that this works correctly.
2. Write the "recursive" case.
    - Make sure all recursive calls go to simpler cases than the one you are writing. Make sure that the simpler cases will eventually get to a base case.
    - Make sure that the general case will work properly if all of the recursive calls work properly.

Recursive Structures as a Collection

We return to graphical examples to see more about ways we can define and use recursive data structures.

One of the earliest programs we wrote was a very simple drawing program that allowed a user to scribble with the mouse.

In writing the scribble program, we realized that a scribble was really just a bunch of tiny line segments. When the user first pressed the mouse to draw, we saved that point as an initial "anchor." As the user dragged the mouse, we drew a tiny line from the anchor point to the new mouse position. Then we used the newer mouse position as our next anchor.

The scribble program we wrote early in the semester was fairly powerful for something so simple. But it is definitely limited. In a more realistic drawing program, the user would undoubtedly want more than the ability to draw scribbles. The user might want to move around the things they've drawn, or might want to change their color, or perhaps even erase them.

Suppose we wish to write a drawing program that allows you to choose a mode: drawing, moving, erasing, coloring (in a particular color: red, blue, yellow, or green). To use the program, choose a mode, and then go to work. To draw, just scribble. To move one of your scribbles, select move, and then drag the desired scribble with the mouse. To color a scribble, select a color mode and then click on the scribble you want to color. Finally, to erase, click on the erase button to get you into the right mode. After that, clicking on the drawing canvas will get rid of scribbles. Those that you created most recently, will be erased first.

Let's consider what we need to do to begin to write the drawing program we've just described. We know that any given scribble is simply a series of line segments. But in this program we need to do more than just draw them. We need to remember them. There are a couple of problems with this: First, to store all of the many line segments in a single scribble, we would need an awful lot of variables. Second, we have no way to predict how many line segments there will be in any given

scribble. Fortunately, the power of recursion can help us here. To see this, consider how we build up a scribble: initially, a scribble is empty. When the user presses the mouse, they signal their intent to draw a scribble, but there's nothing there yet. As soon as a mouse drag is detected, we attach a new line to the existing (empty at first) scribble. With each subsequent drag, we attach a new line segment to the existing scribble.

So a scribble is simply a line attached to a slightly smaller scribble.

This notion should be familiar. We have already seen other examples of objects that are constructed from slightly simpler objects of the same type: a `NestedSquares` object is simply a `FilledRect` with a smaller `NestedSquares` embedded inside of it; a `Scribble` is a `Line` attached to a smaller `Scribble`.

Unlike with `NestedSquares`, we do not construct a `Scribble` all at once – a `Scribble` needs to constructed piece by piece (or, to be more specific, `Line` by `Line`) as the user drags the mouse. To see how this is done, consider the constructor for the `Scribble` class in this example:

See Example: SimplestRecScribbler

To build a scribble, we simply attach a new line to an existing scribble. This is simple, but we have to be a little careful. When the first line is drawn, to what do we attach it? To an empty scribble! To construct an empty scribble, we simply pass to the `Scribble` constructor two parameter values of `null`. The constructor remembers these as the values of its `first` and `rest`, *i.e.*, the first item in the scribble and the rest of the scribble.

Now each time a new line needs to be added to the scribble, *i.e.*, with each drag of the mouse, we construct a newer (bigger) scribble from the new line and the old (current) scribble.

Using our Scribbles

We considered a very simple controller above, so that we could focus on the task of building up a scribble dynamically, as the user moves the mouse. At this point, we can consider how to make use of the fact that we can now remember the current scribble – something we couldn't do in our scribble program at the beginning of the semester. Like `NestedSquares` a `Scribble` is a recursive data structure. Let's consider how we can expand the `Scribble` class to add the ability to move a scribble or to check for containment:

See Example: SimpleRecScribbler

The `move` method should look very familiar. It is nearly identical to `move` from `NestedSquares`. To move a scribble, we simply move the first item in the `Scribble` (a `Line`) and then we move the rest, but only if the `Scribble` is not the empty `Scribble`. If the `Scribble` is an empty one, we do nothing (this is the base case for the recursion). Since checking whether a `Scribble` is empty is useful in a number of different contexts, we write a separate method that checks just this. Recall from above that we created an empty `Scribble` by making its `first` and `rest` variables `null`. Thus checking for emptiness is simply a matter of checking whether `first` is `null`.

To check for containment, we check whether the `Scribble` is empty. If it is, we return `false`

right away (this is the base case!). Otherwise, we check if the point is is contained in the first item in the `Scribble` (another base case!). If so, we return `true` and we're done. If not, we need to check whether the point is contained in some other part of the `Scribble` by invoking the `contains` method on the `rest` of the `Scribble`.

A Stack

Let's leave our scribbles for a moment, and move to stacking wood.

See Example: RecWoodPile

With each click of the mouse on the "Stack 'em" button, a new piece of wood is stacked on the top of the pile. Rather than simply visually changing the picture on the screen, this program has a variable that allows it to remember the stack of wood. Let's examine the `Stack` class that is our template for a "wood pile" object.

You should immediately notice that a stack of wood is very similar to a scribble! It is a recursive data structure: a `Stack` is simply a piece of wood (*i.e.*, a `VisibleImage`) on top of a smaller `Stack`. A stack of wood is constructed but putting together a `VisibleImage` and a smaller stack of wood, as is illustrated in the controller for example. An empty `Stack` is created in the `begin` method. Then each time the user clicks on the "Stack 'em" button, we build a bigger `Stack` out of a new `VisibleImage` and the old (smaller) `Stack`.

Next, let's look more closely at the `Stack` class. As we discussed above, a `Stack` is very similar to `Scribble`, but it is also different, and the difference is reflected in the methods for the two classes. We likely think of a `Scribble`, conceptually, as a single (complex) entity. But we think of the `Stack` as a collection of individual pieces of wood. Id wouldn't make much sense to move the whole stack, for example. But we might want to add to it (which I can by using the constructor), or I might want to take things off of it. The `removeMostRecent` method does just this. It has a single pre-condition – that the stack isn't empty. The method does two things: it modifies the `Stack` so that it no long contains the item that was first; it also returns that item so that the caller of the method can do something with the item (if they choose to). The `removeMostRecent` method makes use of two other methods: `getFirst` and `getRest` that return the values of the instance variables `first` and `rest`, respectively.

While the examples we examined above were quite different, there are some interesting attributes that they share. Both a `Scribble` and a `Stack` can be represented by a recursive data structure. In each case, we used an instance variable to keep track of the most recently added item and another to keep track of the rest. To build either, we started out by building an explicitly empty one and then added to it. We'll see next that this general form of data structure is useful in many more contexts.

Another Example

In considering the `Scribble` and `WoodPile` examples, we oberved that both a scribble and a pile of wood could be represented by classes that are very similar. Both require us to be able to represent a collection of items. In the case of a scribble, we need to remember a collection of `Line`s. In the case of a stack, we need to remember a collection of `VisibleImage`s. In both

cases we need to be able to build the collections dynamically – that is, under the control of the user. In both cases, we started by constructing empty collections. Each time a new item needed to be added, we constructed a new collection from the old one and the new item.

Our next example is to create a recursive data structure to represent a list of balls for an animation. With each click of the mouse, a ball is drawn on the canvas. When the mouse exits the canvas, the balls move, resulting in an animation that looks like a chain reaction. When the mouse enters the canvas, the canvas is cleared and we can start all over.

See Example: ChainReaction

In this case, we need to store a collection of balls. As each new ball is constructed, we need to add it to a data structure. We also need to be able to access the elements of the data structure (in order to make them move). Class `BallList` is used for this purpose.

You should immediately recognize the definition of this class. It is identical in many ways to the `Scribble` and `Stack` classes seen earlier. A `BallList` has two instance variables, `first` and `rest`. The assumption is that a `BallList` will be built up one item at a time. The constructor takes two parameters: one that is a new ball to be added, and a second that is an existing list to which the addition is to be made.

Initially, a user will construct an empty `BallList`. Whenever a new ball is to be added (in `onMousePress` in our example), a new list is constructed by combining a new ball with the existing list.

Now, the reaction part. When we invoke `react` on a `BallList`, it creates a new `ActiveObject` that makes all the balls move a little bit, one at a time.

In `run`, we iterate over the elements of the `ballList`, one at a time until `ballList` is empty. We assign the first element of the list to `nextBall`, move that ball `distanceToMove` pixels to the right using smooth animation, and then step down the list by setting `ballList` to be the rest of the list after the first ball.

Linked Lists: what all these have in common

We've been very careful to note the similarities in the examples we've considered in this series of examples. While scribbles, woodpiles, and balls look very different on the surface, the data structures used to represent them have been strikingly similar. Obviously this is more than just an interesting coincidence. The type of data structure we've been looking at is one that is useful in many contexts. It is called a *singly linked list*.

The name singly linked list might be self-explanatory, but let's say a little more about it. The reason for calling this data structure a list is because it allows us to store a list of items: lines, or wood images, or balls. We call it singly linked because the way it is represented allows us to traverse it in one direction only. To "walk through" the list we always considered `first` first; then we stepped through the list until we reached the end. We never moved backward through the list.

When we build a general-purpose data structure to hold a collection of items, there are certain types of functionality we want them to have. We need to be able to:

- add a new item to the collection
- remove an item from the collection
- tell whether the collection is empty
- get at the individual items in the collection

While we didn't need to implement all of these for all of our examples, we illustrated each of these in at least one context.

Now let's think about that drawing program we talked about near the start of this whole topic. We've already seen how a single scribble can be represented. But what about a collection of scribbles? A collection of scribbles (or a `ScribbleList` as we'll call it) is simply a list of scribbles! How will it be used? We'll want to build an empty one initially. Then each time a scribble is drawn, we'll need ot add it to the collection. Occasionally a user will want to remove one from the list. Finally, we'll need to occasionally get at the individual items in the list – so that the user can select one for moving or for coloring. These are all the types of methods we've already seen illustrated.