

Section 11: Recursively Recursive Recursion
--

Pair up with anyone who is agreeable to pairing up with you, pick the first driver, and let's get to work!

PART I: A Short Recursive Method — Multiplication via Repeated Addition


1. We all learned in grade school that, when dealing with positive integers, multiplication is a short-cut notation for repeated addition. For example, $3 * 7 = 3 + 3 + 3 + 3 + 3 + 3 + 3 = 7 + 7 + 7$.

We can look at this repeated-addition approach recursively. We'll start as we always do, by asking this question:

*What's slightly simpler than ... computing $3 * 7$?*

What's your answer?

2. The next question: *How can you use your answer to construct the answer to the $3 * 7$ question?*
3. Next, express that answer using variables, say m and n , instead of 3 and 7. (That is, write the General Case.)
4. Last question: *What is the Base Case (or Cases) for this problem?* (Remember that the problem description was for positive integers.)
5. Visit the section web page, find the shell of the `Multiplication.java` program, and load it into DrJava.
6. Complete that program so that it uses your Base and General cases to recursively compute and display the product of two positive integers given on the command line.

 **CHECKPOINT 1** Raise your hand. Your SL will come over and see how well your recursive multiplication method works.

PART II: A Recursive Linked-List Method — Printing a List in Reverse Order

1. Find the file `TestSequence.java` on the class web page and load it into DrJava.
2. That file contains: (a) The `LLNode` class that we've used before, (b) the `Test` class (just a renamed (and completed) `Exam` class from last week), and (c) the `TestSequence` class, which is `ExamList` from last week renamed, but with a method named `printReverse()` and a method stub also named `printReverse()`. They are near the top of the file; find them, and review the rest of the code, to remind yourself about the methods.
3. *Question:* Why is it necessary that there be two versions of the `printReverse()` method?

(Continued ...)

4. Your job: Complete the `printReverse()` stub so that it recursively prints the content of the `TestSequence` object's list in reverse order; that is, the content of the last node is printed first, and the content of the first node is printed last. For example, if the list's nodes reference, in order, `Test` objects containing the names "Xavier", "Yule", and "Zblewski", all in section C, the output should be:

```
C:Zblewski
C:Yule
C:Xavier
```

Remember:


- ...to ask yourself "The Question" to help get started, to think of the general case based on your answer to that question, to choose the base case that will stop the general case, and to do all of that before you write any code.
 - ...that `Test` has a `toString()` method.
5. `TestSequence` has a `main()` method, for testing. When you're ready to give your recursive `printReverse()` method a try, compile and run `TestSequence`. Your output should be:

```
Testing printReverse()...
```

```
The first list contains 0 tests: ()
In reverse order:
```

```
The second list contains 1 test: (C:Anderson)
In reverse order:
C:Anderson
```

```
The third list contains 5 tests: (A:Adams)(A:Baker)(A:Cook)(A:Dumerville)(A:Easton)
In reverse order:
A:Easton
A:Dumerville
A:Cook
A:Baker
A:Adams
```

 **CHECKPOINT 2** Raise your hand. Your SL will come over and ask for your answer to the question, and see how you did writing your method.

PART III: A Graphical Recursive Method — Overlapping Shapes

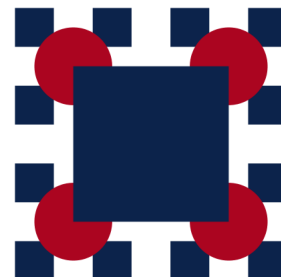
1. Recursion is a great tool for creating interesting graphics. Consider the picture sequence:



Degree = 1



Degree = 2

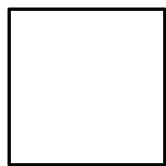


Degree = 3

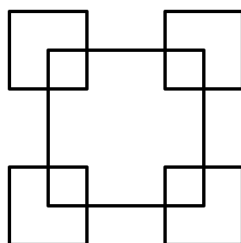
If you finish this part, you'll have created a method capable of producing those pictures, and in even more detail. But, we'll tackle something a bit easier first; read on!

(Continued ...)

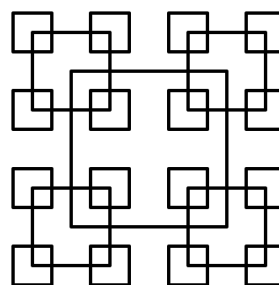
- Find `OverlappingShapes.java` on the class web page and load it into DrJava.
- Our first goal is to complete the `draw()` method in that program so that it creates these images, where the degree value is input to the program on the command line:



Degree = 1



Degree = 2



Degree = 3


Examine the progression and be sure that you could explain how the “Degree = 4” picture would differ from the “Degree = 3” picture.

- Complete `draw()` so that it produces the overlapping square diagrams shown above, using recursion, subject to the following:
 - The base case is $d = 0$; nothing is drawn.
 - In the general case, a bigger square and four smaller squares are drawn, with the bigger square being drawn after the smaller ones. (This ordering is what will (eventually) provide the overlapping effect seen in the first set of pictures.)
 - The biggest square is centered at $(0.5, 0.5)$ and has a radius of 0.25 ; these are the initial values passed to `draw()`.
 - Each smaller square is half the width/height of its bigger square, is centered on one of the bigger square’s corners, and is drawn by a recursive call to `draw()`. Thus, the general case of `draw()` includes four recursive calls.

NOTE: `OverlappingShapes.java` also contains a method named `fourShapes()` that demonstrates how to draw circles and squares using the `StdDraw` class from the `stdlib.jar` package. Uncommenting the call in the main method will allow you to see it, if you think that seeing it will help.

Make it happen! When you run the program using degrees of one, two, and three, and see those interlocking square pictures, you’re ready to make the images more interesting.


- To produce the images shown at the top of this part, you need to make the shape drawing in the general case just a little more complex. Instead of drawing empty squares, modify `draw()` to do this:
 - When the degree is an even number, draw a filled red circle, using the `UA_RED` color predefined for you.
 - When the degree is an odd number, draw a filled blue square, using the `UA_BLUE` color.
- When you’re seeing the correct color images, you’re done!

 **CHECKPOINT 3** Raise your hand. Your SL will come over and marvel at your recursive artwork.

(Continued ...)

PART IV: Clean Up!

1. Log out of your computer; pick up your papers, writing implements, cell phones, trash, etc.; push in your chair(s).

 **CHECKPOINT 4** Raise your hand. Your SL will come over and ask you for your recommendation for a good brand of disinfecting wipes.

“PART V”: Extra Practice!

Finished the above early and need something to do to fill the rest of the period? Struggled to finish the above and know you need more practice? Then consider trying these!

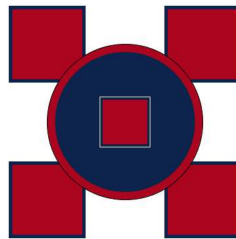
1. At the bottom of `TestSequence`'s main method is some testing code for a method named `copyReverse()`. `copyReverse()` creates and returns a copy of the list, as a new `TestSequence` object, but in reverse order. As such, it can be thought of as an extension of the `printReverse()` idea.

Write `copyReverse()`. To test it, uncomment that testing code, and recompile and rerun `TestSequence`.

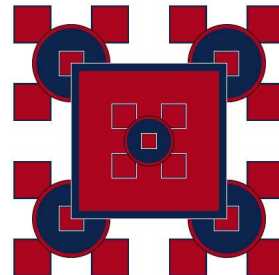
2. By adding some ‘fancy’ color borders and edging, and just one additional recursive call, to your final version of `draw()` from Part III, you can create a version that produces this sequence:



Degree = 1



Degree = 2



Degree = 3

Can you figure out how?

3. For lots of recursion problems for practice, visit CodingBat.com/java. You will see two sections titled **Recursion-1** and **Recursion-2**. Stick to those in the first group. If you can do them all without help, you'll know that you really understand recursion well!