Recursion 1 – Developing Recursive Algorithms

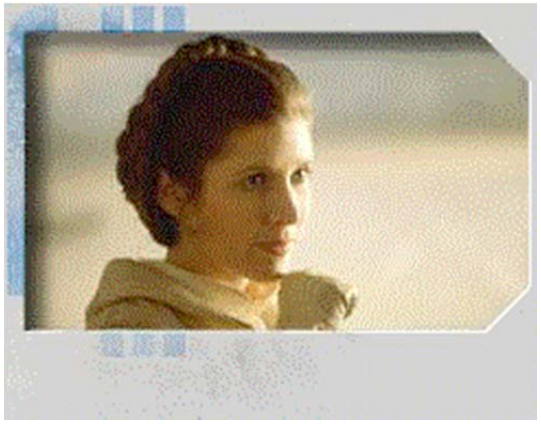
Preview of Coming Attractions

In this unit be sure to look for

- recursion
- recursive structure
- 5 step process for developing recursive algorithms
- base case, base values

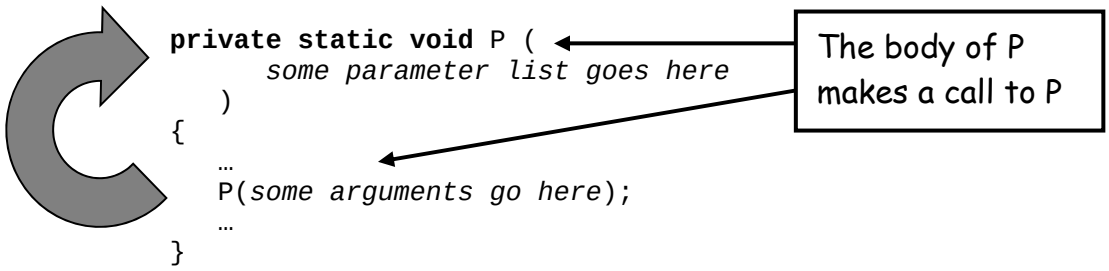
The Force Is With You

One of the most totally awesome features of programming languages goes by the name of recursion. You'll experience this "awesomeness" by writing short, simple, and understandable algorithms that solve very complicated problems. You might even find yourself giddy with joy or feeling a sense of amusement. One thing is for sure — you'll know that the force is with you.



What Is It?

The word recursion derives from the word recur, and one of the meanings of recur is to occur, or appear, again. In programming languages, *recursion* refers to methods where the method body makes one or more calls to the method itself. Here's what this looks like:



So, a call to the method P executes the body of P, which also calls P, so that calls to P recur.

Feel the Force

In order to be an effective user of recursion when developing recursive algorithms, you'll need to do two things:

1. ***See recursive structure in the values of an method's parameters.***
2. ***See how to leverage this recursive structure into a recursive algorithm.***

In short, you'll need to see, ***in your mind's eye***, things that you may have never seen before. (By seeing with your *mind's eye*, we mean that seeing is not necessarily seeing literally with your eye. The "seeing" can be just in your mind.)

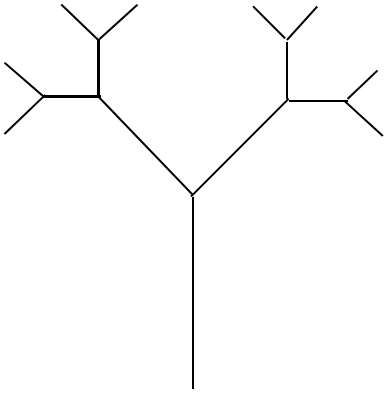
What Is Seeing Recursive Structure?

Seeing recursive structure is ***not looking at*** something. Rather, it is ***looking into*** something and seeing, within, smaller things, just like the thing at which you are looking. For example, imagine in your mind a big, sweet, juicy onion. Imagine “peeling off” just the outer layer of that onion. Now what do you see? Do you see another onion, only smaller? In your mind, you have looked *into* an onion and have seen a smaller onion; you have seen the recursive structure of onions— onions within onions.

Normally software does not perform computations on onions. But there are plenty of examples of computational things that do have recursive structure. We'll look at three examples. Each example is presented as a picture, and is suggestive of the type of picture you might want to form *in your mind's eye*.

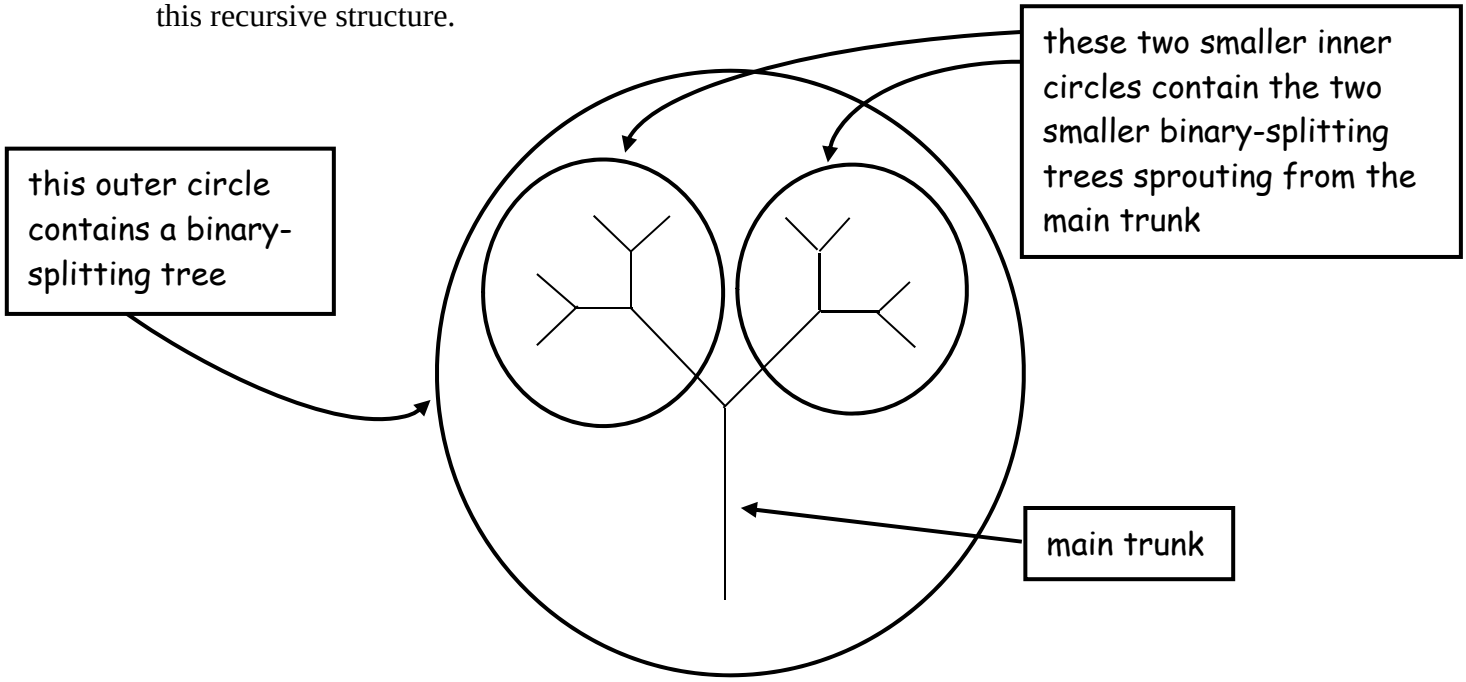
Binary-Splitting Trees

Here's a picture of an exotic plant form that we'll call a binary-splitting tree.



A Binary-Splitting Tree— An Exotic Plant

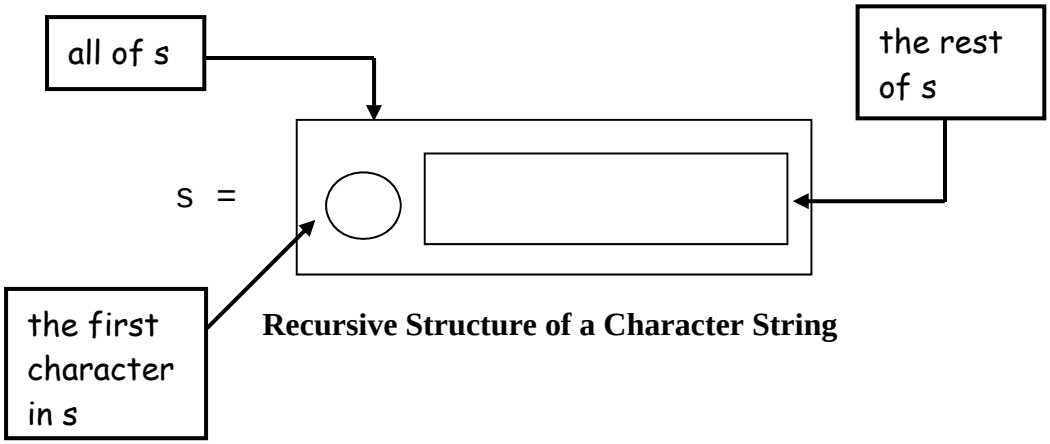
If we look *into* this tree, we can see recursive structure. There is a main trunk, with two branches sprouting out to the right and left of the main trunk, each at a 45-degree angle to the main trunk. But these two branches are just the main trunks for two *smaller* binary-splitting trees, where the main trunks for these two smaller trees are each half as long as the main trunk from which they sprouted, and so on. The next figure helps us to capture this recursive structure.



Recursive Structure of a Binary-Splitting Tree

Character Strings

Consider a character string named `s`:



Recursive Structure of a Character String

The larger, outer box in this picture contains a character string. If we "peel off" the first (left-most) character of the character string, indicated by the circle, we are left with the rest of the character string, which is just another character string. The smaller, inner box indicates this smaller character string. So, if we look *into* a character string, we can see a smaller character string. In this way, we can see recursive structure in character strings—character strings within character strings.

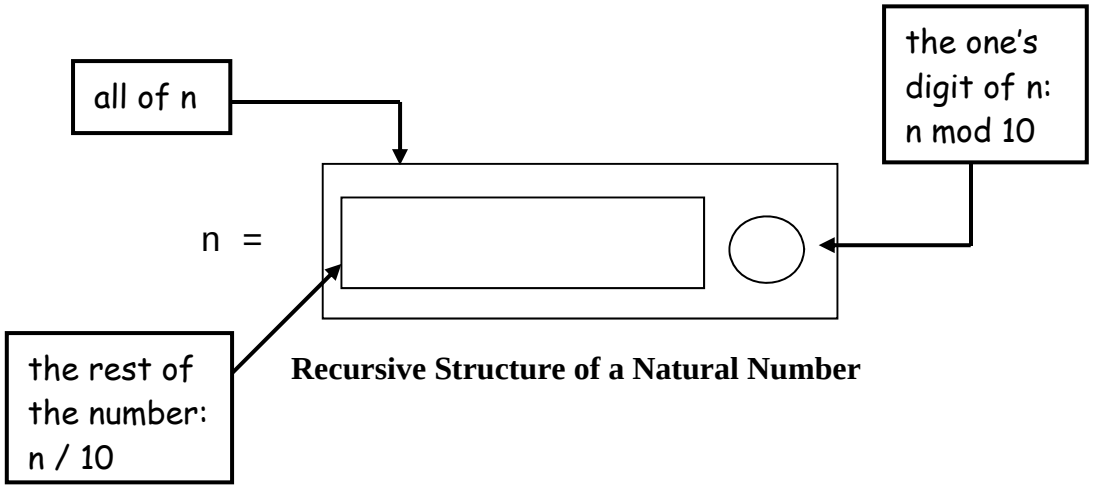
Still Awake?

R1-1. If *s* is a `String` variable, how would you separate (or extract) the first character and the rest of the `String`? Write Java statements that print the first character and the rest of the `String` to a given `SimpleWriter` variable named *out*.

R1-2. Think of another way to see recursive structure in character strings. Draw a circle/box picture illustrating your idea.

Natural Numbers

Consider a natural number named n:



Recursive Structure of a Natural Number

The larger, outer box in this picture contains a natural number. If we "peel off" the last (right-most) digit of the natural number, indicated by the circle, we are left with the rest of the natural number, which is just another natural number. The smaller, inner box indicates this smaller natural number. So, if we look *into* a natural number, we can see a smaller natural number. In this way, we can see recursive structure in natural numbers—natural numbers within natural numbers.

Still Awake?

R1-3. If *n* is a `NaturalNumber` variable, how would you separate (or extract) the one's digit and the rest of the number? Write Java statement that print the one's digit and the rest of the number to a given `SimpleWriter` variable named *out*.

What Does It Mean to Leverage Recursive Structure?

We've just considered three examples of seeing *into* things to reveal recursive structure. You might still be wondering, "Why bother?" Well, the only reason to look for recursive structure is to determine whether seeing recursive structure can help to develop nice algorithms to accomplish computational tasks. Let's consider three examples to get some understanding of what it might mean to use a view of recursive structure to develop an algorithm.

Drawing a Binary-Splitting Tree

Suppose we need to implement a method to draw a binary-splitting tree. Further, let's suppose the method uses a couple of parameters, such as "drawing pen" that is located at some screen location and pointing in a specific direction, and a length (in screen units) for the main trunk of the tree to be drawn. We have seen that a binary-splitting tree consists of a main trunk and two smaller binary-splitting trees that sprout from the main trunk. Can we leverage this recursive structure to develop an algorithm to draw a binary-splitting tree? Well, ask yourself this question: ***If the drawing method could get a helper to display the two, smaller binary-splitting trees, could the drawing method take advantage of this generous offer to make drawing the entire tree an easy task?*** It's a no brainer! The drawing method would draw the main trunk, locate the drawing pen at the base of the main trunk of the smaller right-sprouting tree, point the drawing pen in the proper direction, and ask the helper to draw the smaller right-sprouting binary-splitting tree. When the helper finishes, the drawing method would then locate the drawing pen at the base of the main trunk of the smaller left-sprouting tree, point the drawing pen in the proper direction, and ask the helper to draw the smaller left-sprouting binary-splitting tree. As soon as the helper finishes, we're all done!

In this little exercise, we asked the question: ***If the drawing method could get a helper to display the two, smaller binary-splitting trees, could the drawing method take advantage of this generous offer to make drawing the entire tree an easy task?*** With just a little thought, the answer is YES! Because the answer is yes, we can now proceed knowing that our particular view of recursive structure is helpful. (In case you are

wondering, when the recursive algorithm is developed, the "helper" will turn out to be a recursive call to the very algorithm we are developing. That's fun!) On the other hand, if we could *not* see how to use the generous offer to make drawing a tree easy, then we either abandon the idea of using recursion in a solution, or we look for another view of recursive structure that might be more helpful.

Still Awake?

R1-4. Using the ideas just discussed, give a skeleton of a recursive body for a method to draw a binary-splitting tree. Remember, the "helper" is replaced with recursive calls to the drawing method. Do you see the need for a third parameter in addition to the drawing pen and the length of the main trunk?

Reversing a Character String

Suppose we need to implement a method, called `reversedString`, which returns the reverse of a given character string, e.g., given `"abcd"`, it returns `"dcba"`. We have seen that a character string consists of a first character followed by the smaller, rest-of-the character string. How can we leverage this recursive structure into an algorithm for `reversedString`? Well, ask yourself this question: ***If the reversing method could get a helper to reverse the smaller character string, could the reversing method take advantage of this generous offer to make reversing the entire character string an easy task?*** Again, it's a no brainer! `reversedString` could remove the first character, ask the helper to reverse the rest of the character string, and, when the helper is done, add the removed first character to the right end of the reversed character string. That's it, `reversedString` is all done.

In this little exercise, we asked the question: ***If the reversing method could get a helper to reverse the smaller character string, could the reversing method take advantage of this generous offer to make reversing the entire character string an easy task?*** With just a little thought, the answer is YES! Because the answer is yes, we can now proceed knowing that our particular view of recursive structure is helpful.

Incrementing a Natural Number

Suppose we need to implement a method to increment a natural number. We have seen that after removing the one's digit, we are left with a smaller, rest-of-the number. Can we leverage this recursive structure to develop an algorithm to increment a natural number? Well, ask yourself this question: ***If the increment method could get a helper to increment the smaller number, could the increment method take advantage of this generous offer to make incrementing the original number an easy task?*** Once again, it's a no brainer! The display method would remove the one's digit from the natural number, add 1 to the digit, check whether this results in a carry, and if it does, ask the helper to increment the smaller number, and, when the helper is finished, put back the updated one's digit. That's it!

In this little exercise, we asked the question: ***If the increment method could get a helper to increment the smaller number, could the increment method take advantage of this generous offer to make incrementing the original number an easy task?*** With just a little thought, the answer is YES! Because the answer is yes, we can now proceed knowing that our particular view of recursive structure is helpful.

Still Awake?

R1-5. Using the ideas just discussed, give a skeleton of a recursive body for a method to increment a natural number. Remember, the "helper" is replaced with recursive calls to the increment method.

Replacing the Helper With a Recursive Call— What's Up With That?

We just considered three little exercises that involved thinking up algorithms for methods, where the algorithms made use of helpers. By replacing calls to the helpers with recursive calls (calls to the same method), our helper-assisted methods become real methods. With just a little thought, the idea of replacing helpers with recursive calls makes a lot of sense.

Consider the drawing method for arbitrary binary-splitting trees. The helper was used to draw the two smaller binary-splitting trees sprouting from the main trunk. But, ***the drawing method itself handles the task of drawing arbitrary binary-splitting trees***, so why not ask the drawing method itself to draw the two smaller binary-splitting trees? When we do this, the result is a recursive method body that makes two recursive calls to the method itself. The only thing different is that in the recursive calls, the binary-splitting trees to be drawn are smaller.

Consider the reversing method for arbitrary character strings. The helper was used to reverse the smaller character string obtained by removing the left-most character of the original character string. But, ***the reversing method itself handles the task of reversing an arbitrary character string***, so why not ask the reversing method itself to reverse the smaller character string? When we do this, the result is a recursive method body that makes one recursive call to the method itself. The only thing different is that in the recursive call, the character string to be reversed is smaller.

Still Awake?

R1-6. Provide a similar justification for replacing the helper, in the natural number increment method, with a recursive call to the increment algorithm itself.

A Process for Developing Recursive Algorithms

Now that you have a sense of the force, it's time to do some serious training. This training will help you use recursion as effectively as possible. Training involves mastering a five-step process for developing recursive algorithms for methods. So, ***assume that your task is to implement a method and that you intend to use recursion to get it done***.

A Process for Developing Recursive Method Bodies

1. Use pictures to visualize appropriate recursive structure for the incoming values of the method's parameters.
2. Verify that the visualized recursive structure can be leveraged into an implementation for the method.
3. Use pictures to visualize a recursive implementation.
4. Write a skeleton for the method body describing informally the recursive implementation visualized in step 3.
5. Gradually refine the skeleton into a method body.

In this process, we are taking the phrase "seeing recursive structure" literally, by actually drawing, in step 1, a picture of recursive structure. Please remember that as you gain familiarity with recursion, you might take the phrase less literally.

An Example

Let's put the process to work on the following method:

```
/**
 * Reverses a String.
 *
 * @ensures <pre>
 * {@code reversedString = rev(s)}
 * </pre>
 */
private static String reversedString(String s) {. . .}
```

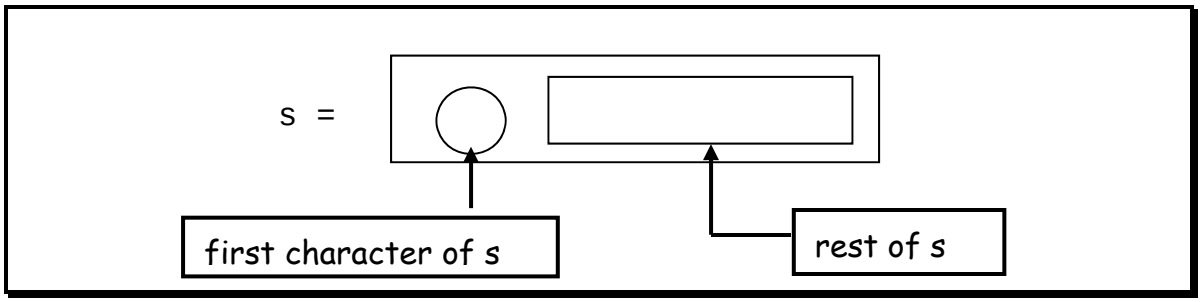
(The little math function ***rev*** indicates the reverse of a string. For example, ***rev***`("1234")="4321"`.) Our job is to implement `reversedString` and ***we want to use recursion to get the job done***.

Still Awake?

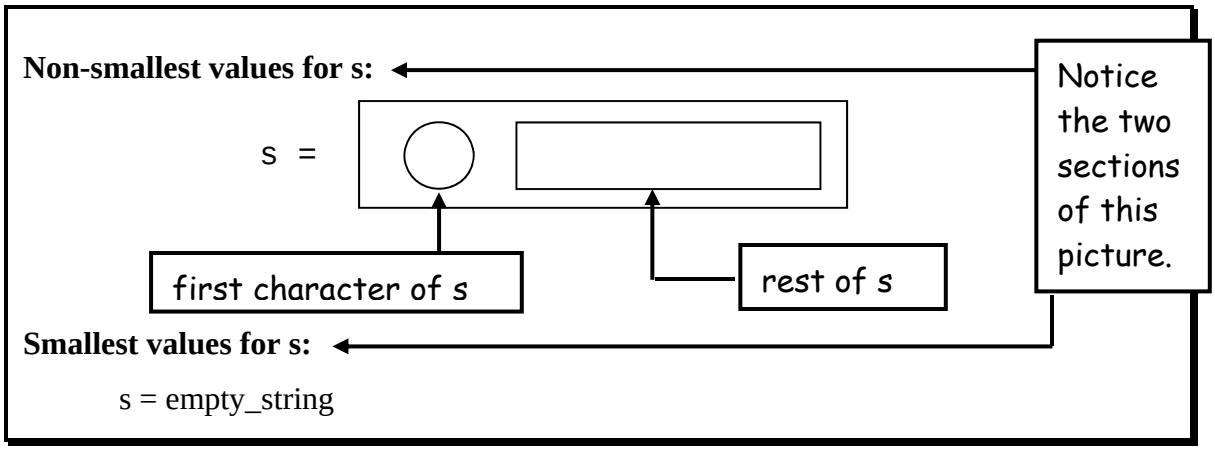
R1-7. Before proceeding, see if you can give a skeleton for a recursive method body for `reversedString`, based on the ideas discussed earlier.

Step 1 — Visualize Recursive Structure

`reversedString` has single parameter `s` of type `String`. How can we see into a character string and discover recursive structure? Let's try our earlier idea and view the incoming value of `s` as:



This looks pretty good, except that the incoming value of `s` may be the empty string. In this case, the picture is pretty silly because there is no first character (to fill the circle) in `s`, because there are no characters in `s` at all. So, let's be just a bit more careful with our visualization of the incoming value of `s`:

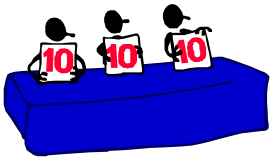


Often, smallest values are referred to as *base values* or *base-case values*. Notice the picture has two sections, one for smallest and one for non-smallest values. This will always be the case for all pictures for step 1.

Step 2 — Verify That Leveraging is Possible

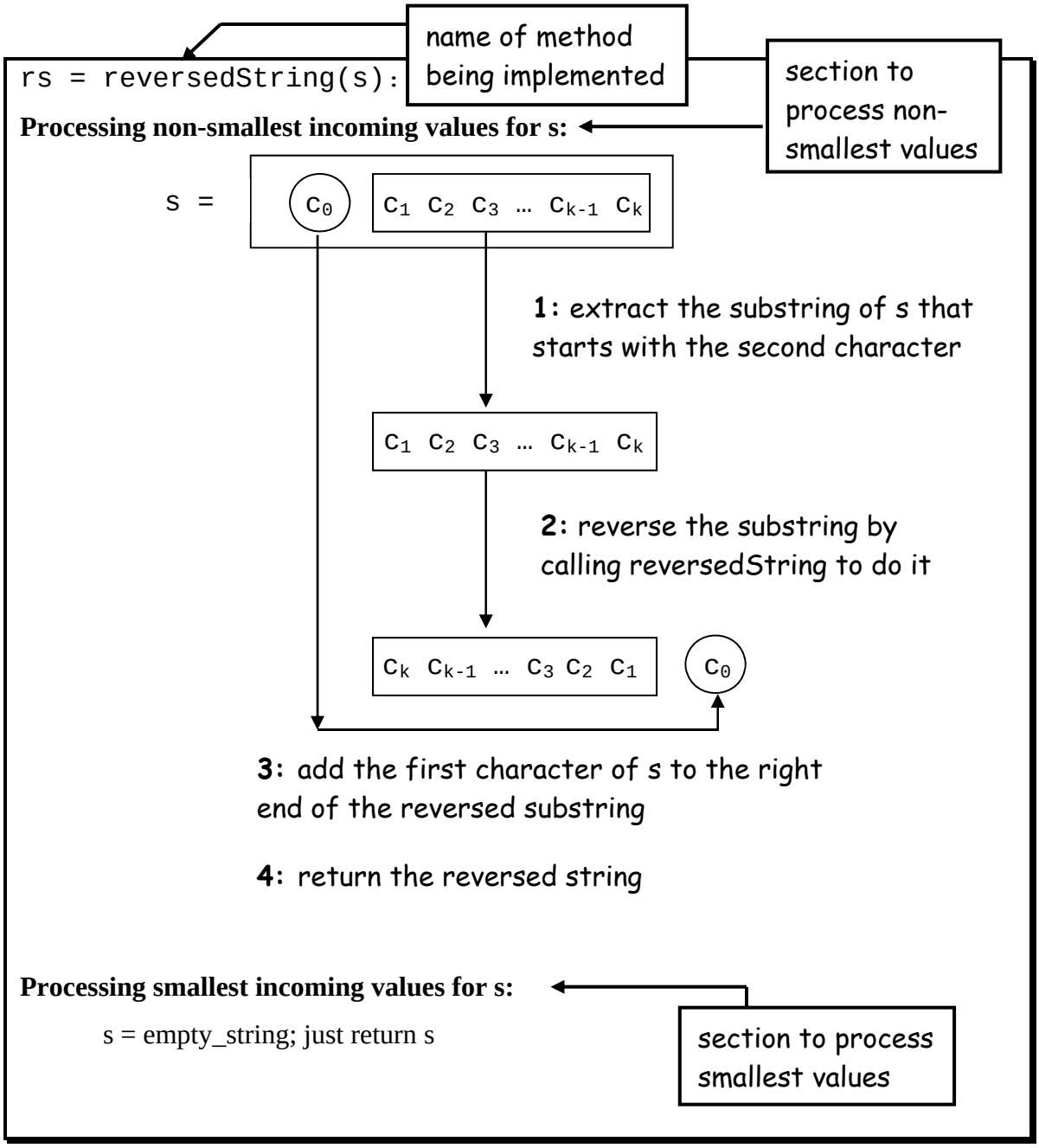
This is a crucial step. If things do not go well here, then we must head back to step 1 and try again. We can move on to step 3 only after our confidence is certain.

How do we get our confidence? Just *ask and answer* the question we asked earlier: ***If the reversing method could get a helper to reverse the smaller character string, could the reversing method take advantage of this generous offer to make reversing the entire character string an easy task?*** As before, `reversedString` could remove the first character, call the helper to reverse the rest of the character string, and then add the removed first character to the right end of the reversed character string. This looks very promising, so we are confident of being on the right track. Let's move on.



Step 3 — Visualize a Recursive Implementation

Here's a visualization of our answer to the question from step 2. Notice that `reversedString` is used as the helper for itself.



Let's see what's in this picture:

1. In the upper left corner, there's the method call `reversedString(s)`. In these pictures, *always* include a method call for the method being implemented. Because `reversedString` returns the reversed character string, we show the returned value being assigned to a new variable.

2. Next comes the processing of *non-smallest values*. The process pictured here reflects our ideas from step 2. To draw this section, *start* with a picture of non-smallest values from step 1. *Annotate* the picture with *actions*. Normally, this will involve at least three actions:
   2.1. ***There should be an action or actions showing how input values (the outer box) are transformed to obtain smaller input values (the inner box).*** In the picture, this is the action labeled with 1: extract the substring of `s` that starts with the second character.
   2.2. ***There should be an action or actions annotated with explicit indications of one or more recursive calls to the method being implemented.*** These calls apply the method to smaller input values (the inner box). In the picture, action 2 includes calling `reversedString` to reverse the substring of `s`.
   2.3. ***There should be some indication of how the solution for smaller input values (the inner box) is used to arrive at a solution for original input values (the outer box).*** In the picture, this is the action labeled 3 that adds the first character of `s` to the right end of the reversed substring.
3. Last is a section describing the processing of *smallest input values*. This is a separate section because smallest values are normally processed differently than non-smallest values.

Step 4 — Write a Skeleton

Here's a skeleton that captures the ideas from step 3.

```
reversedString(s) {
   if (s is not the empty string) {
      // process non-smallest values
      extract the substring of s that starts with the
         second character;
      reverse the substring by calling reversedString to do it;
      add the first character of s to the right end of the
         reversed substring;
   } else {
      // process smallest values
      return s;
   }
}
```

The form or structure of this skeleton is a cliché — almost every recursive method body will have the same structure:

```
someMethod(…) {
   if (input does not have a smallest value) {
      // process non-smallest values
      // code to process non-smallest values goes here
   } else {
      // process smallest values
      // code to process smallest values goes here
   }
}
```

Pretty simple, eh? By the way, processing smallest values is often referred to as processing the base case.

Step 5 — Refine the Skeleton Into a Method Body

Working straight from the skeleton, here's a body for the `reversedString` method:

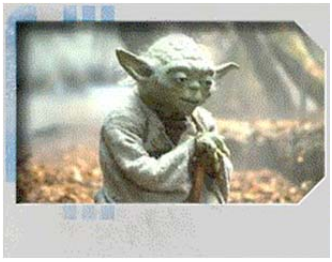
```
private static String reversedString(String s) {
   if (s.length() > 0) {
      // process non-smallest values
      String sub = s.substring(1);
      String revSub = reversedString(sub);
      String result = revSub + s.charAt(0);
      return result;
   } else {
      // nothing to do for smallest values, except returning s
      return s;
   }
}
```

Still Awake?

R1-8. In an earlier Still Awake? exercise, you described another way of seeing recursive structure in text strings. Complete the 5-step process for developing recursive methods bodies to arrive at a second implementation of `reversedString`, based on your earlier idea.

Training Completed

Well, that's the end of training camp for developing method bodies that use recursion. Go forth and practice now you must. The force to use remember.



Unit Wrap-up

Comments appearing in the next figure ARE FUNDAMENTALLY IMPORTANT. Make sure you understand them completely. Put them into an accessible, long-term part of your memory!

If your task is to develop an algorithm to implement a method, and if you intend to use recursion in the method body, then you must

- Identify recursive structure in the data to be processed.
- Understand how to arrive at simple algorithm for the task at hand by making use of the identified recursive structure.