Developing a recursive method

We demonstrate how to develop a recursive method. The development is based on the four steps used to verify the correctness of a recursive method.

1. Have a precise, clear specification

We develop a function with a String parameter `s` that returns `s` but with adjacent duplicates removed. Our first task is to write a specification. We do so to the right. The example makes clear what is mean by "adjacent duplicates".

```
/** = s with adjacent duplicates removed.
  * e.g. for s = "abbcccdeaaa", the
  * answer is "abcdea". */
public static String remDups(String s)
```

Examples help the reader verify their understanding of the spec.

2. Determine and implement the base cases

The base cases generally satisfy two properties: (1) They are the smallest possible values for the parameters. (2) The answer for the base cases can be determined without resorting to recursion.

When asked what the smallest String value is, many tyros answer "a one-character string". They forget about the empty string! Don't do that. The shortest string is the empty string.

Both the empty string and a one-character string do not contain duplicates, so we take care of these two base cases with the following:

```
if (s.length() <= 1) return s;
```

Step 3. Determine and implement the recursive cases. Any string of at least two characters may have duplicates. Let's write such a string `s` in a way that makes those two characters explicit: `s`[0] + `s`[1] + `s`[2..].

If `s`[0] = `s`[1], there are adjacent duplicates, and `s`[0] can be deleted. Therefore, the result can be written as

`s`[1..], with adjacent duplicates removed.

Look at the similarity of this phrase and the specification of `remDups`:

= `s` with adjacent duplicates removed.

This tells us that the result is the value of the recursive call `remDups` (`s`[1]) and we can handle this case using the Java statement:

```
if (s.charAt(0) == s.charAt(1)) return remDups (s.substring(1));
```

This example illustrates the main strategy in implementing recursive cases:

Strategy: Write the answer using a phrase that is the same as the spec except that parameters may be replaced by other expressions. Then, replace the phrase using a recursive call.

Here, `s` in the spec is replaced by `s`[1..] in the result.

Doing this would be impossible if we didn't have a good, precise spec!

In the case that s[0] != s[1], the result is s[0] + (s[1..] with adjacent duplicates removed), and we implement this result with a recursive call:

```
return s.charAt(0) + remDups (s.substring(1));
```

We end up with the method shown to the right. Note the comment we put in at the beginning of the recursive cases. The assertion helps us remember what is known at that point.

```
/** = s with adjacent duplicates removed.
  * e.g. for s = "abbcccdeaaa", the
  * answer is "abcdea". */
public static String remDups(String s) {
   if (s.length() <= 1) return s;

   // s = s[0] + s[1] + s[2..]
   if (s.charAt(0) == s.charAt(1))
      return remDups (s.substring(1));
      return s.charAt(0) +
             remDups (s.substring(1));
}
```

Step 4. Check that the recursion terminates. Termination of recursion is easy here. Each recursive call has an argument that is one character shorter than parameter `s`, so the maximum depth of recursion is `s.length()`.