Recursion: Why It Works
Question Considered Before

• **How should you think about** recursion so you can use it to develop elegant recursive methods to solve certain problems?

• **Answer**: Pretend there is a *FreeLunch* class with a method that has the same *contract* as the code you’re trying to write (but it works only for *smaller* problems)
Question Considered Now

• *Why* do those recursive methods work?
Question Considered Only Later

• *How* do those recursive methods work?
  – Don’t worry; we will come back to this
  – If you *start* by insisting on knowing the answer to this question, you may never be fully capable of developing elegant recursive solutions to problems!
Example #1

```java
private static String reversedString(String s) {
    if (s.length() == 0) {
        return s;
    } else {
        String sub = s.substring(1);
        String revSub = reversedString(sub);
        String result = revSub + s.charAt(0);
        return result;
    }
}
```
Example #1

```java
private static String reversedString(String s) {
    if (s.length() == 0) {
        return s;
    } else {
        String sub = s.substring(1);
        String revSub = reversedString(sub);
        String result = revSub + s.charAt(0);
        return result;
    }
}
```

There is no reason to declare the variable result, only to return it in the next statement.
private static String reversedString(String s) {
    if (s.length() == 0) {
        return s;
    } else {
        String sub = s.substring(1);
        String revSub = reversedString(sub);
        return revSub + s.charAt(0);
    }
}
Confidence-Building Approach

• We can make an intuitive confidence-building argument that the code is correct (i.e., correctly implements its contract)

• Consider the size metric that allows you to argue that a smaller problem is being solved by each recursive call in your code
  – In this example, recall we used as a measure of problem size $|s|$, the length of the value of parameter $s$
First, a Smallest Problem

• First, consider each *smallest* problem according to that metric
  – In this example, there is exactly one smallest problem: $|s| = 0$, i.e., $s = < >$

• Convince yourself that your code works correctly for each smallest problem
  – Trace the code (though you could also execute it as if testing it on these cases)
Trace Code With $|s| = 0$

<table>
<thead>
<tr>
<th>if $s.length() == 0$</th>
<th>s = &quot;&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>return $s$;</td>
<td>s = &quot;&quot;</td>
</tr>
</tbody>
</table>
Trace Code With $|s| = 0$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s = &quot;&quot;$</td>
<td></td>
</tr>
<tr>
<td>if (s.length() == 0) {</td>
<td></td>
</tr>
<tr>
<td>$s = &quot;&quot;$</td>
<td></td>
</tr>
<tr>
<td>return s;</td>
<td></td>
</tr>
</tbody>
</table>

The code in this case returns the value \\
"" 
(since $s = \"\")
and this satisfies the postcondition
$reversedString = \text{rev}(s)$
so it works when $|s| = 0$. 
Then, a Next-Smallest Problem

• Now, consider a *next-smallest* problem according to that metric
  – In this example, there are many next-smallest problems, but all have $|s| = 1$, e.g., these include $s = "a", s = "X", s = "\%", etc.

• Convince yourself that your code works correctly for each of these next-smallest problems
  – Maybe one such problem is convincing...
Trace Code With $|s| = 1$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s = &quot;X&quot;$</td>
<td></td>
</tr>
<tr>
<td><strong>if</strong> ($s$.length() == 0) { ... } <strong>else</strong> {</td>
<td></td>
</tr>
<tr>
<td>$s = &quot;X&quot;$</td>
<td></td>
</tr>
<tr>
<td>String sub = $s$.substring(1);</td>
<td></td>
</tr>
</tbody>
</table>
| $s = "X"$  
  sub = "" |   |
| String revSub = reversedString(sub); |   |
| $s = "X"$  
  sub = ""  
  revSub = "" |   |
| **return** revSub + $s$.charAt(0); |   |
How do we know what this recursive call does? We just convinced ourselves it satisfies its contract in this case because $|\text{sub}| = 0$ in this recursive call.
The code in this case returns the value "X" (since `revSub = ""` and `s = "X"`) and this satisfies the postcondition `reversedString = rev(s)` so it works when `|s| = 1`.

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>s = &quot;X&quot;</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>sub = &quot;&quot;</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>revSub = &quot;&quot;</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>return revSub + s.charAt(0);</code></td>
</tr>
</tbody>
</table>
Trace Code With $|s| = 1$

If $s.length() == 0$ { ... } else {
    String sub = s.substring(1);
    sub = "";
    String revSub = reversedString(sub);
    return revSub + s.charAt(0);
}

Note that concluding “it works when $|s| = 1$” demands that we generalize from the one case we traced/tested, realizing that there was nothing at all special about the assumption $s = "X"$. 

$s = "X"$

$s = "X"$

$s = "X"$

$s = "X"$

$s = "X"$

$s = "X"$

$s = "X"$

$s = "X"$

$s = "X"$

Note that concluding “it works when $|s| = 1$” demands that we generalize from the one case we traced/tested, realizing that there was nothing at all special about the assumption $s = "X"$. 

$s = "X"$. 
Then, a Next-Smallest Problem

• Now, consider a *next-smallest* problem according to that metric
  – In this example, there are many next-smallest problems, but all have $|s| = 2$, e.g., these include $s = "rx"$, $s = "PU"$, etc.

• Convince yourself that your code works correctly for each of these next-smallest problems
  – Maybe one such problem is convincing...
trace code with \( | s | = 2 \)

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (s.length() == 0) { ... } else {</code></td>
<td><code>s = &quot;PU&quot;</code></td>
</tr>
<tr>
<td><code>String sub = s.substring(1);</code></td>
<td><code>s = &quot;PU&quot;</code></td>
</tr>
</tbody>
</table>
| `String revSub = reversedString(sub);` | `s = "PU"`  
  `sub = "U"`  
  `revSub = "U"` |
| `return revSub + s.charAt(0);` | `s = "PU"`  
  `sub = "U"`  
  `revSub = "U"` |
How do we know what this recursive call does? We just convinced ourselves it satisfies its contract in this case because $|\text{sub}| = 1$ in this recursive call.

<table>
<thead>
<tr>
<th>$s$</th>
<th>sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PU&quot;</td>
<td>&quot;U&quot;</td>
</tr>
</tbody>
</table>

```java
String revSub = reversedString(sub);

return revSub + s.charAt(0);
```
The code in this case returns the value

"UP"

\(revSub = \text{"U"} \text{ and } s = \text{"PU"}\)

and this satisfies the postcondition

\(\text{reversedString} = \text{rev}(s)\)

so it seems to work when \(|s| = 2\).

| s | = 2 |
|---|
| s = "PU" |
| s = "PU" |
| s = "PU" |
| s = "PU" |
| sub = "U" |
| sub = "U" |
| revSub = "U" |

```
String revSub = reversedString(sub);

return revSub + s.charAt(0);
```
Trace Code With $|\mathbf{s}| = 2$

<table>
<thead>
<tr>
<th>$\mathbf{s} = &quot;PU&quot;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (s.length() == 0) { ... } else {</td>
</tr>
<tr>
<td>$\mathbf{s} = &quot;PU&quot;$</td>
</tr>
<tr>
<td>String sub = s.substring(1);</td>
</tr>
<tr>
<td>$\mathbf{s} = &quot;PU&quot;$</td>
</tr>
<tr>
<td>sub = &quot;U&quot;</td>
</tr>
<tr>
<td>$\mathbf{sub} = &quot;U&quot;$</td>
</tr>
<tr>
<td>revSub = reversedString(sub);</td>
</tr>
<tr>
<td>$\mathbf{revSub} = &quot;U&quot;$</td>
</tr>
<tr>
<td>return revSub + s.charAt(0);</td>
</tr>
</tbody>
</table>

Note that concluding “it works when $|\mathbf{s}| = 2$” demands that we generalize from the one case we traced/tested, realizing that there was nothing at all special about the assumption $\mathbf{s} = "PU"$. 
Then, a Next-Smallest Problem

- Now, consider a next-smallest problem according to that metric
  - In this example, there are many next-smallest problems, but all have $|s| = 3$, e.g., these include $s = "cse")$, $s = "OSU")$, etc.

- Convince yourself that your code works correctly for each of these next-smallest problems
  - Maybe one such problem is convincing...
Trace Code With $|s| = 3$

<table>
<thead>
<tr>
<th>Code</th>
<th>s = &quot;OSU&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (s.length() == 0) { ... } else {</td>
<td></td>
</tr>
<tr>
<td>String sub = s.substring(1);</td>
<td>s = &quot;OSU&quot;</td>
</tr>
<tr>
<td>String revSub = reversedString(sub);</td>
<td></td>
</tr>
<tr>
<td>return revSub + s.charAt(0);</td>
<td>s = &quot;OSU&quot; sub = &quot;SU&quot; revSub = &quot;US&quot;</td>
</tr>
</tbody>
</table>
How do we know what this recursive call does? We just convinced ourselves it satisfies its contract in this case because $|\text{sub}| = 2$ in this recursive call.
The code in this case returns the value 
"USO" 
(revSub = "US" and s = "OSU")
and this satisfies the postcondition 
reversedString = rev(s) 
so it seems to work when |s| = 3.
Trace Code With $|s| = 3$

```java
if (s.length() == 0) { ... } else {
    String sub = s.substring(1);
    sub = "SU"
    String revSub = reversedString(sub);
    return revSub + s.charAt(0);
}
```

Note that concluding “it works when $|s| = 3$” demands that we generalize from the one case we traced/tested, realizing that there was nothing at all special about the assumption $s = "OSU"$. 
And So On...

- You should see that this reasoning process could continue long enough to reach any finite integer length, with the conclusion that the code works for any value of $s$
  - Because $s$ must have a finite length; why?
Proof by Induction

• A formal version of this argument follows the proof technique known as *mathematical induction*
  – Recursion and induction are entirely parallel concepts
Proof by Contradiction

• Another formal proof technique known as *proof by contradiction* can also be used
  — “Suppose the code does *not* work for some $s$. Then there must be a shortest $s$ for which it does not work. So, assume $|s| = n$. Now let’s show that this assumption leads to the conclusion that the code must not work for some string of length $n-1$. This is a contradiction. Hence, there cannot be any such $s$ for which the code does not work.”
Proof by Contradiction

- Another formal proof technique known as proof by contradiction can also be used.
  - “Suppose the code does not work for some $s$. Then there must be a shortest $s$ for which it does not work. Let’s show that this assumption leads to the conclusion that the code must not work for some string of length $n–1$. This is a contradiction. Hence, there cannot be any such $s$ for which the code does not work.”

Some people find this kind of proof easier to understand than induction, but it does not seem to have a simplified intuitive basis like induction does.
```java
private static void increment (NaturalNumber n) {
    int onesDigit = n.divideBy10();
    onesDigit++;
    if (onesDigit == 10) {
        onesDigit = 0;
        increment(n);
    }
    n.multiplyBy10(onesDigit);
}
```
First, a Smallest Problem

• First, consider each **smallest** problem according to the size metric: the value of the parameter $n$
  – In this example, there is exactly one smallest problem: $n = 0$

• Convince yourself that your code works correctly for each smallest problem
  – Trace through the code with $n = 0$
Then, a Next-Smallest Problem

• Next, consider a *next-smallest* problem according to that metric
  – In this example, there is exactly one next-smallest problem: $n = 1$

• Convince yourself that your code works correctly for each of these next-smallest problems
  • Trace through the code with $n = 1$
And So On...

- On this example, the intuitive confidence-building argument may be slightly more convincing because every value of parameter $n$ is covered directly in one of the steps.
- But it takes 9 steps before anything interesting even happens!
Conclusion

• The purpose of the confidence-building method is (as its name suggests) to give you confidence that the code works
  – A nice feature is that it suggests some test cases, should you decide to run the code on the computer rather than tracing it manually
  – However, you still need to have traced it — albeit perhaps only mentally — in order to have written the code in the first place!
Conclusion

The purpose of the confidence-building method (as its name suggests) is to give you confidence that the code works – A nice feature is that it suggests some test cases should you decide to run the code on the computer rather than tracing it manually – However, you still need to have traced it — albeit perhaps only mentally — in order to have written the code in the first place!

You might be surprised how many people who should know better seem not to have noticed this rather obvious conclusion!