Recursion: How 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 Before

• **Why** do those recursive methods work?
• **Answer**: Following the “confidence-building” approach, you can argue as follows:
  – Does it work on all “smallest” cases?
Question Considered Before

• **Why** do those recursive methods work?
• **Answer**: Following the “confidence-building” approach, you can argue as follows:
  – Does it work on all “smallest” cases? ✓
  – Does it work on all “next smallest” cases?
Question Considered Before

• **Why** do those recursive methods work?

• **Answer**: Following the “confidence-building” approach, you can argue as follows:
  – Does it work on all “smallest” cases? ✓
  – Does it work on all “next smallest” cases? ✓
  – Does it work on all “next smallest” cases?
Question Considered Before

• **Why** do those recursive methods work?
• **Answer**: Following the “confidence-building” approach, you can argue as follows:
  – Does it work on all “smallest” cases? ✓
  – Does it work on all “next smallest” cases? ✓
  – Does it work on all “next smallest” cases? ✓
  – … (Formally, proof by mathematical induction)
Question Considered Now

• **How** do those recursive methods work?
  – As promised, we have come back to this, but we continue to advise...
  – If you insist on *thinking about recursion* this way (rather than simply satiating your curiosity about how it works), you may never be fully capable of developing elegant recursive solutions to problems!
Example

```java
private static String reversedString(String s) {
    if (s.length() == 0) {
        return s;
    } else {
        String sub = s.substring(1);
        String rSub = reversedString(sub);
        return rSub + s.charAt(0);
    }
}
```
Trace `reversedListString("OSU")`

<table>
<thead>
<tr>
<th>if (s.length() == 0) { ... } else {</th>
</tr>
</thead>
<tbody>
<tr>
<td>String sub = s.substring(1);</td>
</tr>
<tr>
<td>String rSub = reversedListString(sub);</td>
</tr>
<tr>
<td>return rSub + s.charAt(0);</td>
</tr>
</tbody>
</table>

```
s = "OSU"
```

```
s = "OSU"
```

```
s = "OSU"
sub = "SU"
```

```
s = "OSU"
```

```
return rSub + s.charAt(0);```

7 January 2019

OSU CSE
Trace `reversedList(String("OSU"))`

<table>
<thead>
<tr>
<th>s = &quot;OSU&quot;</th>
<th>if (s.length() == 0) { ... } else {</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = &quot;OSU&quot;</td>
<td>String sub = s.substring(1);</td>
</tr>
<tr>
<td>s = &quot;OSU&quot;</td>
<td>String rSub = reversedString(sub);</td>
</tr>
</tbody>
</table>

Question: This is a recursive call, so how does it work?
Trace `reversedList("OSU")`

| if (s.length() == 0) { ... } else { | s = "OSU" |
| String sub = s.substring(1); | s = "OSU" |
| String rSub = reversedString(sub); | s = "SU" |

Answer:
Exactly like this one, and every other call!
How Every Call Works

• First, the tracing table for the code making the call is *suspended* and that tracing table is *pushed onto the runtime stack*
  – The runtime stack, often called simply “the stack”, is effectively just a stack of tracing tables (think `Stack<TracingTable>`, each partially filled in with the results of the code in that tracing table as executed *so far*)
How Every Call Works

• A new tracing table is created, containing the code for the method body being called
• The argument values are copied from the suspended tracing table into the formal parameters to start the new tracing table
• Execution in the new tracing table continues until it calls a method...
The currently executing tracing table gets to here ...

<table>
<thead>
<tr>
<th>String(&quot;OSU&quot;)</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></td>
</tr>
<tr>
<td>String rSub = reversedString(sub);</td>
<td></td>
</tr>
<tr>
<td>return rSub + s.charAt(0);</td>
<td></td>
</tr>
</tbody>
</table>
... and the (top of the) stack of suspended tables is here.

```java
String s = "OSU"

if (s.length() == 0) {
    ...
} else {

    String sub = s.substring(1);

    String rSub = reversedString(sub);

    return rSub + s.charAt(0);
```

Trace `reversedString("OSU")`... and the (top of the) stack of suspended tables is here.
This call suspends the current tracing table ...

```
String reversedString("OSU")

|                      | $s = \"OSU\"$
|----------------------|-----------------------------
| if (s.length() == 0) { ... } else { | |
|                      | |
| String sub = s.substring(1); | |
|                      | |
| String rSub = reversedString(sub); | |
|                      | |
| return rSub + s.charAt(0); | |
```
... the suspended tracing table is \textit{pushed} ...
... and the tracing table for length begins.

```java
String("OSU")
```

<table>
<thead>
<tr>
<th>this = &quot;OSU&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>this = &quot;OSU&quot;</td>
</tr>
<tr>
<td>length = 3</td>
</tr>
</tbody>
</table>

/*
 * Body for length method
 * from class String; we
 * do not have this, so how
 * do we know what it does?
 * We look at its contract!
 * When it finishes, we know
 * it has not changed this
 * (it could not even if it
 * wanted to), and it returns
 * the length of this.
 */

```java
Trace reversedString("OSU")
```
How Every Return Works

• When the currently executing tracing table reaches a `return` statement, or for a `void` method falls off the end of the body, the results of the call are reflected in the tracing table on the `top of the stack`

• That tracing table is `popped off the stack` and it becomes the currently executing tracing table, resuming execution from the point where it was suspended
When this call returns ...

```java
String("OSU")
```

<table>
<thead>
<tr>
<th>this = &quot;OSU&quot;</th>
</tr>
</thead>
</table>
| /*
  * Body for length method
  * from class String; we
  * do not have this, so how
  * do we know what it does?
  * We look at its contract!
  * When it finishes executing
  * it has not changed this
  * (it could not even if it
  * wanted to), and it returns
  * the length of this
  */ |
| this = "OSU" |
| length = 3 |
... its results are reflected in the calling table ...

```java
String reversedString("OSU")
```
String("OSU")

... and that table is **popped** to resume execution.

| if (s.length() == 0) { ... } else { |
| String sub = s.substring(1); |
| String rSub = reversedString(sub); |
| return rSub + s.charAt(0); |
String("OSU")

<table>
<thead>
<tr>
<th>s = &quot;OSU&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (s.length() == 0) { ... } else {</td>
</tr>
<tr>
<td>String sub = s.substring(1);</td>
</tr>
<tr>
<td>String rSub = reversedString(sub);</td>
</tr>
<tr>
<td>return rSub + s.charAt(0);</td>
</tr>
</tbody>
</table>

Execution continues to the next call ...
```java
String s = "OSU"
if (s.length() == 0) {
  ...
} else {
  s = "OSU"
  String sub = s.substring(1);
  String rSub = reversedString(sub);
  return rSub + s.charAt(0);
}
```

... and when that call returns, to the next call ...
String("OSU")

... which is a **recursive** call! But it is nothing special.

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

Trace reversedString("OSU")... which is a recursive call!
But it is nothing special.
The current tracing table is suspended and pushed ...

```java
String reversedString("OSU")
```
String("OSU")

if (s.length() == 0) { ...
} else {

String sub = s.substring(1);

String rSub = reversedString(sub);

return rSub + s.charAt(0);
... begins *with its own* variables and values.

```java
String s = "SU";

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

Trace reversedString("OSU")... begins with its own variables and values.
Soon, this tracing table reaches a recursive call!
String("OSU")

The current tracing table is suspended and pushed ...
String("OSU")

<table>
<thead>
<tr>
<th>s = &quot;U&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (s.length() == 0) { ...</td>
</tr>
<tr>
<td>} else</td>
</tr>
</tbody>
</table>

String sub = s.substring(1);

String rSub = reversedString(sub);

return rSub + s.charAt(0);
String("OSU")

... begins with its own variables and values.

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

Trace reversedString("OSU") begins with its own variables and values.
Soon, this tracing table reaches a recursive call!
The current tracing table is suspended and pushed ...

```java
String reversedString(String str) {
    String s = str;
    String sSub = new String();
    for (int i = 0; i < str.length(); i++) {
        sSub = sSub + str.charAt(i);
        s = s.substring(0, s.length() - 1);
    }
    return s + sSub;
}
```
```java
String reversedString("OSU")
```

... and a new table for the body of the called method ...

```java
String s = ""
if (s.length() == 0) {
    return s;
}
```
... begins with its own variables and values.
Soon, this tracing table returns ...

```java
String reversedString("OSU")

s ="
if (s.length() == 0) {
    s ="
return s;
```
... its results are reflected in the calling location ...
... and that table is *popped* to resume execution.

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

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

Soon, this tracing table returns ...
... its results are reflected in the calling location ...
String(“OSU”)

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

... and that table is popped to resume execution.
String("OSU")

if (s.length() == 0) {
    ...
} else {
    String sub = s.substring(1);
    String rSub = reversedString(sub);
    return rSub + s.charAt(0);
}

Soon, this tracing table returns ...
... its results are reflected in the calling location ...
String("OSU")

<table>
<thead>
<tr>
<th></th>
<th>s = &quot;OSU&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (s.length() == 0) {</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>String sub = s.substring(1);</td>
<td>s = &quot;OSU&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s = &quot;OSU&quot;</td>
</tr>
<tr>
<td></td>
<td>sub = &quot;SU&quot;</td>
</tr>
<tr>
<td>String rSub =</td>
<td></td>
</tr>
<tr>
<td>reversedString(sub);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>return rSub + s.charAt(0);</td>
<td></td>
</tr>
</tbody>
</table>
Soon, this tracing table returns ...

```java
String reversedString(String s) {
    if (s.length() == 0) { ... }
    else {
        String sub = s.substring(1);
        String rSub = reversedString(sub);
        return rSub + s.charAt(0);
    }
}
```
Finally!

• The value returned to the original calling program is the string "USO"
  – Phew!
  – And it is even correct: the result of reversing the string "OSU" is the string "USO"
Conclusion

• Each call to a method—whether recursive or not—effectively results in the creation of a new tracing table containing the body of the called method

• Each tracing table has its own variables:
  – Its own formal parameters
  – Its own local variables
Conclusion

• If you really think you can reason about recursive code by mentally executing this kind of a series of events to check your thinking, then ... you’re deluding yourself
Conclusion

• If you really think you can reason about recursive code by mentally executing this kind of a series of events to check your thinking, then ... you’re deluding yourself.

And if you don’t believe it yet, try mentally executing this way for code that makes multiple recursive calls from each tracing table.