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 sating your curiosity about how it works), you may never be fully capable of developing elegant recursive solutions to problems!

Example

```
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 `reversedString("OSU")`

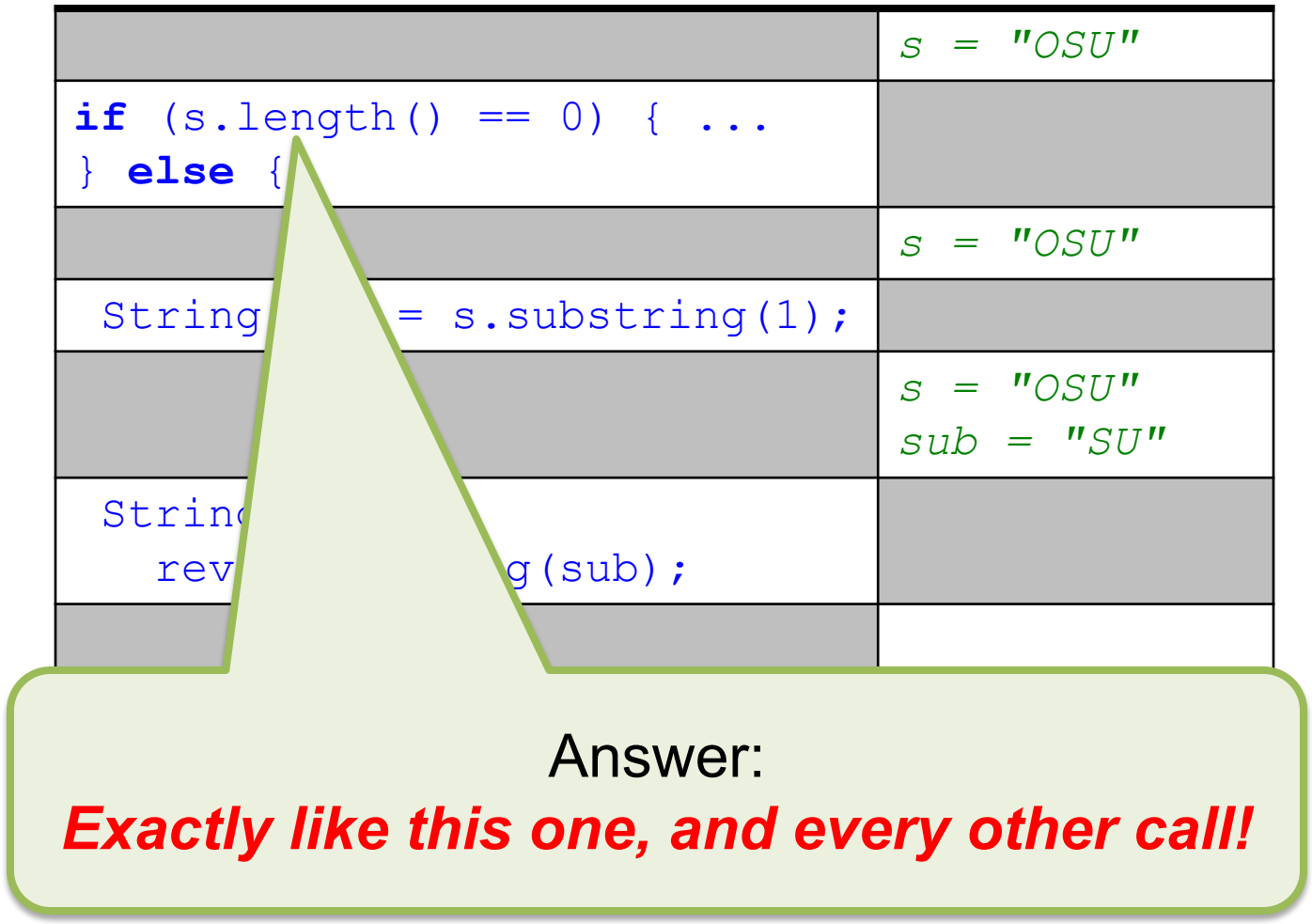
	s = "OSU"
if (s.length() == 0) { ... } **else** {	
	s = "OSU"
String sub = s.substring(1);	
	s = "OSU" *sub = "SU"*
String rSub = reversedString(sub);	
return rSub + s.charAt(0);	

Trace `reversedString("OSU")`

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

Question:
This is a recursive call, so how does it work?

Trace `reversedString("OSU")`



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 ...

reversedString("OSU")

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

... and the (top of the) stack of suspended tables is here.

String("OSU")

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

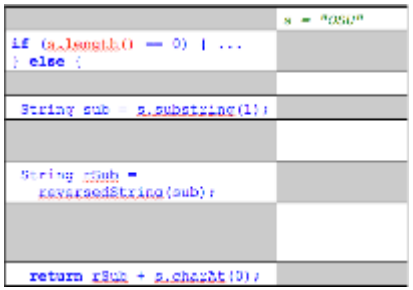
This call suspends the current tracing table ...

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 ***pushed*** ...

`String("OSU")`



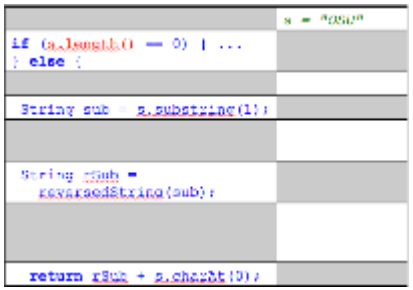
... and the tracing table for `length` begins.

`String("OSU")`

	this *= "OSU"*

`/*`
` * 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`**`.`
` */`	

	this *= "OSU"* *length = 3*

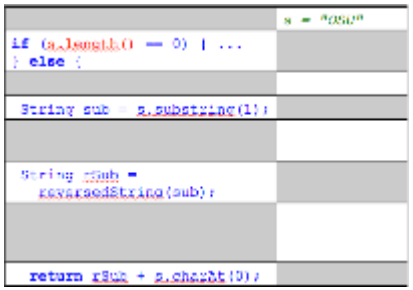


How Every Return Works

- 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 ...

String("OSU")



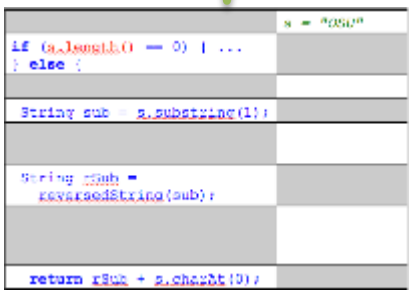
```
/*
 * Body f      gth method
 * from cla    ring; we
 * do not hav    s, so how
 * do we know w    it does?
 * We look at its    tract!
 * When it finishes    ecuting
 * it has not changed    his
 * (it could not even i    it
 * wanted to), and it re    rns
 * the length of this
 */
```

this = "OSU"

this = "OSU"
length = 3

... its results are reflected in the calling table ...

`String("OSU")`



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

reversedString("OSU")

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

Execution continues to the next call ...

String("OSU")

	s = "OSU"
if (s.le() == 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")

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

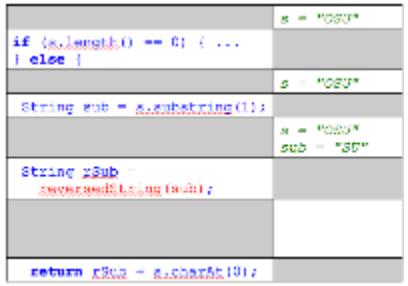
... which is a ***recursive*** call!
But it is nothing special.

reversedString("OSU")

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

The current tracing table is suspended and pushed ...

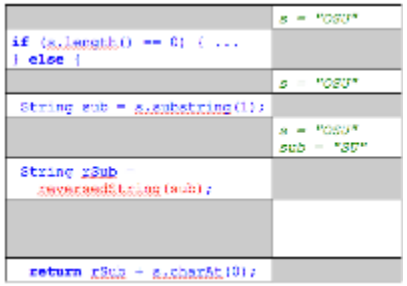
`String("OSU")`



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

reversedString("OSU")

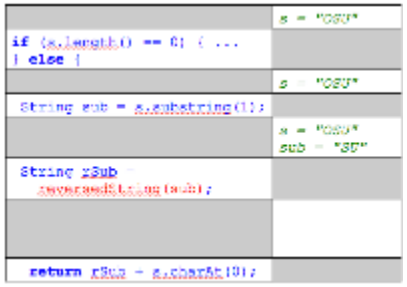
	s = "SU"
if (s.length() == 0) { ... } **else** {	
String sub = s.substring(1);	
String rSub = reversedString(sub);	
return rSub + s.charAt(0);	



... begins ***with its own*** variables and values.

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

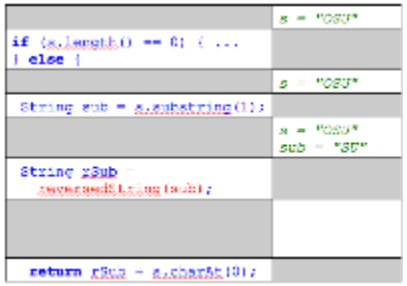
	s = "SU"
if (s.length() == 0) { ... } **else** {	
String sub = s.substring(1);	
String rSub = reversedString(sub);	
return rSub + s.charAt(0);	



Soon, this tracing table reaches a recursive call!

reversedString("OSU")

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



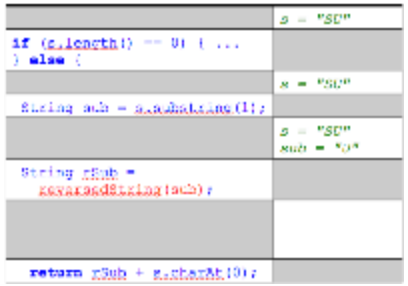
The current tracing table is suspended and pushed ...

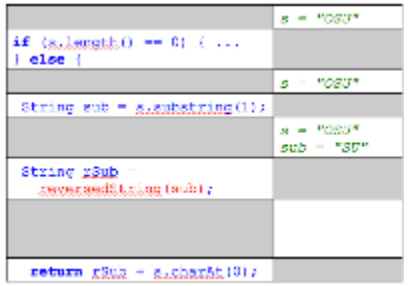
`String("OSU")`

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

reversedString("OSU")

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

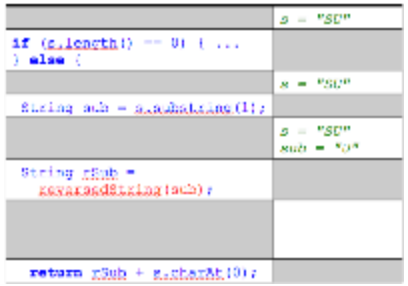


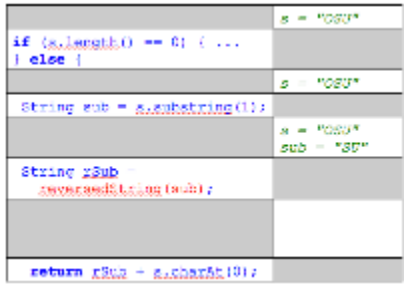


... begins ***with its own*** variables and values.

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

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





Soon, this tracing table reaches a recursive call!

String("OSU")

	s = "U"
if gth() == 0) { ... } **el**	
	s = "U"
Stri sub = s.substring(1);	
	s = "U" *sub = ""*
String rSub = reversedString(sub);	
return rSub + s.charAt(0);	

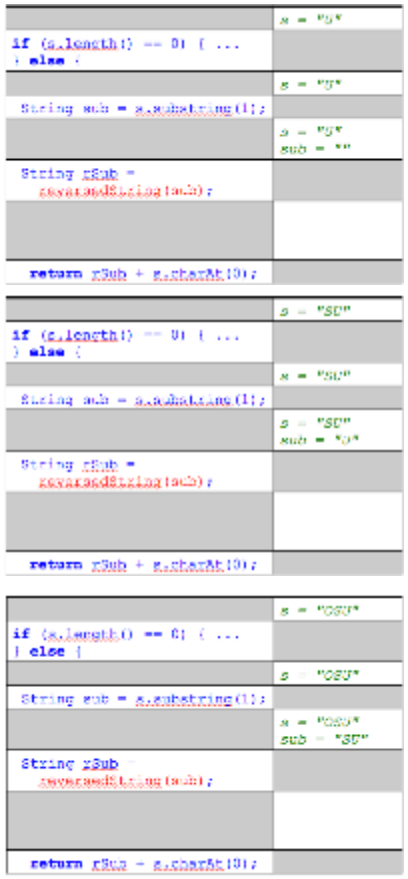
The current tracing table is suspended and pushed ...

`String("OSU")`

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

reverseString("OSU")

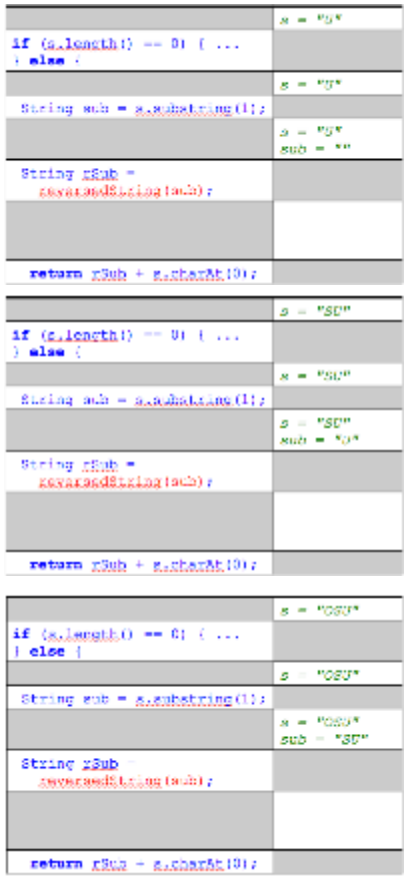
	s = ""
`if (s.length() == 0) {`	
`return s;`	



... begins ***with its own*** variables and values.

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

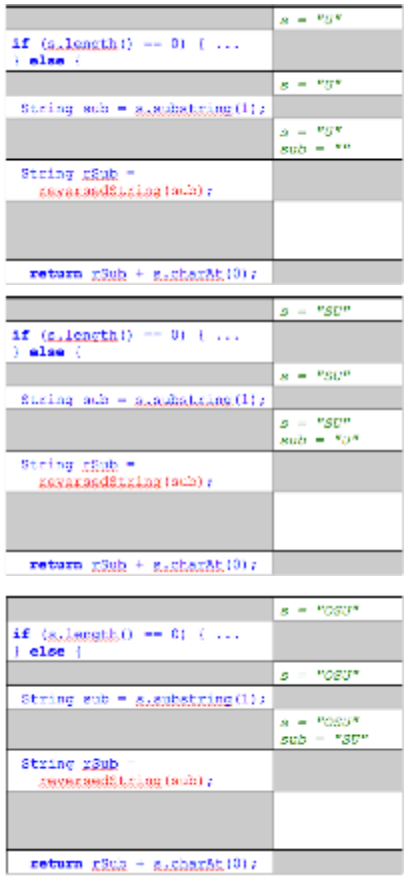
	s = ""
if (s.length() == 0) {	
return s;	



Soon, this tracing table returns ...

reverseString("OSU")

	s = ""
if (ngth() == 0) {	
	s = ""
return s;	



... its results are reflected in the calling location ...

`String("OSU")`

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

reversedString("OSU")

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

Soon, this tracing table returns ...

String("OSU")

	s = "U"
if gth() == 0) { ... } e	
	s = "U"
Str ub = s.substring(1);	
	s = "U" sub = ""
Stri rSub = rev sedString(sub);	
	s = "U" sub = "" rSub = ""
return rSub + s.charAt(0);	

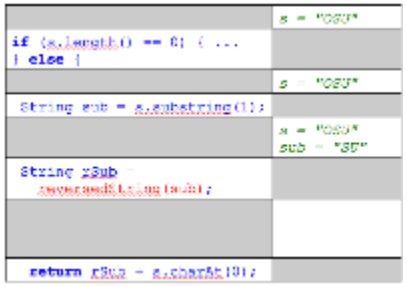
... its results are reflected in the calling location ...

`String("OSU")`

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

reversedString("OSU")

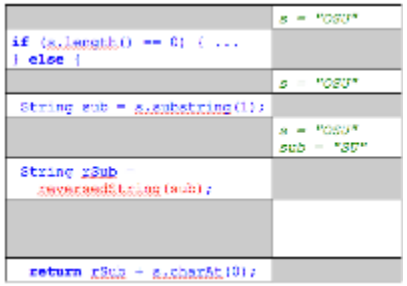
	s = "SU"
if (s.length() == 0) { ... } **else** {	
	s = "SU"
String sub = s.substring(1);	
	s = "SU" *sub = "U"*
String rSub = reversedString(sub);	
return rSub + s.charAt(0);	



Soon, this tracing table returns ...

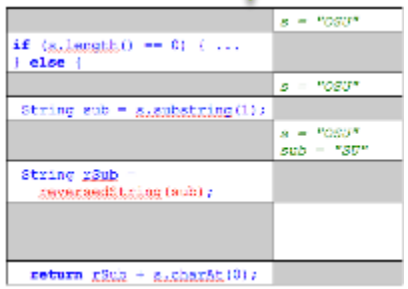
reverseString("OSU")

	s = "SU"
if (s.length() == 0) { ... } e...	
	s = "SU"
Str... sub = s.substring(1);	
	s = "SU" sub = "U"
Stri... rSub = rev...sedString(sub);	
	s = "SU" sub = "U" rSub = "U"
return rSub + s.charAt(0);	



... its results are reflected in the calling location ...

`String("OSU")`



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

reversedString("OSU")

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

Soon, this tracing table returns ...

String("OSU")

	s = "OSU"
if gth() == 0) { ... } e	
	s = "OSU"
Str ub = s.substring(1);	
	s = "OSU" *sub = "SU"*
Stri rSub = rev sedString(sub);	
	s = "OSU" *sub = "SU"* *rSub = "US"*
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.