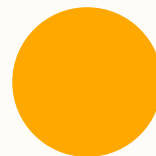




CS 1181

Week Nine

Reese Hatfield

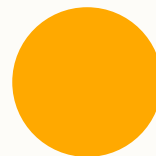


0



Maps and More

Reese Hatfield



0



Map ADT

- Key Value Pairs
- Associate one value to another
- Really fast get() operations

Map ADT
+put(key, value)
+ contains(key)
+ clear()
+ get(key): Value
+ remove(key)





Map ADT

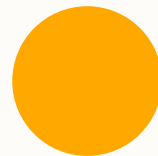
- Maps show up *all* the time
- People → Favorite Color
- Countries → Capitals
- Let's practice using them a bit more
 - Letter occurrence counter

countLetters(targetLetter, String)





Callstacks and Recursion



0



Callstack

- We've briefly talked about the callstack
 - Stack ADT
 - Method call \Rightarrow `Stack.push()`
 - Return \Rightarrow `Stack.pop()`
- Track the execution of our programs with this idea





Callstack

```
public static void main(String[] args){  
    int a = 6;  
    int b = 8  
  
    int result = doMath(a, b)  
  
    System.out.print(result)  
  
}
```

Programs Call Stack





Callstack

Main gets
called

 `public static void main(String[] args){`

`int a = 6;`

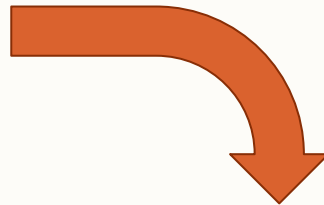
`int b = 8`

`int result = doMath(a, b)`

`System.out.print(result)`

`}`

Put main on
the stack



`$ main()`

Programs Call Stack





Callstack

```
public static void main(String[] args){
```



```
    int a = 6;
```

```
    int b = 8
```

```
    int result = doMath(a, b)
```

```
    System.out.print(result)
```

```
}
```

\$ main()

Programs Call Stack





Callstack

```
public static void main(String[] args){
```

```
    int a = 6;
```



```
    int b = 8
```

```
    int result = doMath(a, b)
```

```
    System.out.print(result)
```

```
}
```

\$ main()

Programs Call Stack





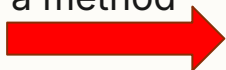
Callstack

```
public static void main(String[] args){
```

```
    int a = 6;
```

```
    int b = 8
```

Need to call
a method



```
    int result = doMath(a, b)
```

```
    System.out.print(result)
```

```
}
```


\$ main()

Programs Call Stack





Callstack



```
public static int doMath(int a, int b){  
    int mathResult;
```

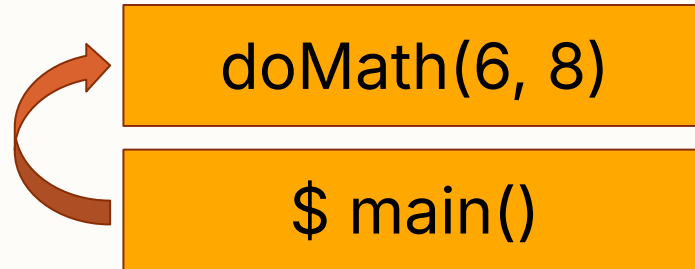
```
    int halfOfA = divide(a, 2)
```

```
    mathResult = mult(halfOfA, b)
```

```
    return mathResult
```

```
}
```

Put doMath()
on the stack



Programs Call Stack





Callstack

```
public static int doMath(int a, int b){
```



```
    int mathResult;
```

```
    int halfOfA = divide(a, 2)
```

```
    mathResult = mult(halfOfA, b)
```

```
    return mathResult
```

```
}
```

doMath(6, 8)

\$ main()

Programs Call Stack





Callstack

```
public static int doMath(int a, int b){  
    int mathResult;
```

Need to call
a method



```
    int halfOfA = divide(a, 2)
```

```
    mathResult = mult(halfOfA, b)
```

```
    return mathResult
```

```
}
```

doMath(6, 8)


\$ main()

Programs Call Stack



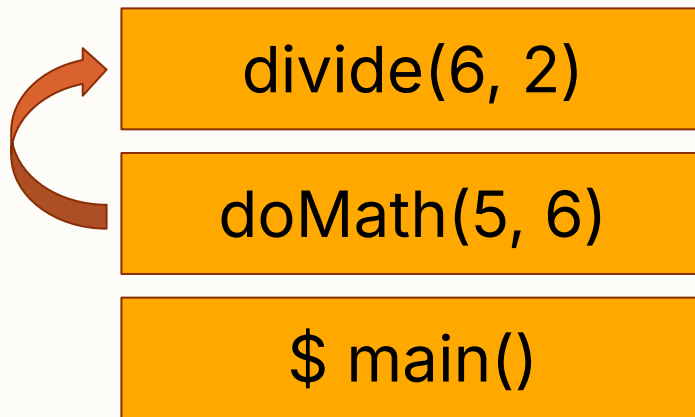


Callstack



```
public static int divide(int n, int d){  
    int result = n / d;  
  
    return result;  
}
```

Put divide() on
the stack



Programs Call Stack





Callstack

```
public static int divide(int n, int d){  
    int result = n / d;  
  
    return result;  
}
```



divide(6, 2)

doMath(5, 6)

\$ main()

Programs Call Stack





Callstack

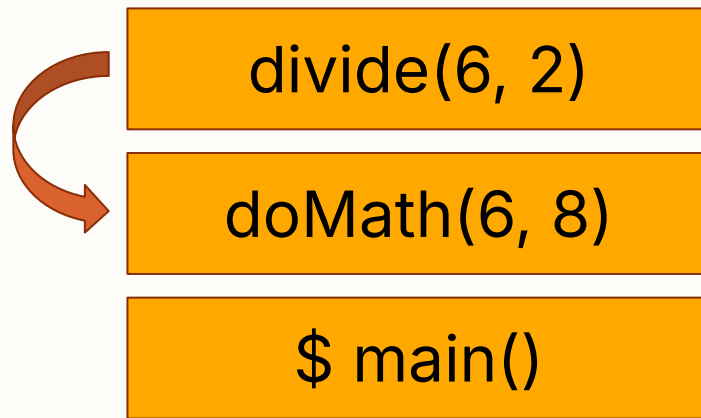
```
public static int divide(int n, int d){  
    int result = n / d;
```

Return ⇒
stack pop



```
    return result;  
}
```

Return right back to where we
left off in doMath



Programs Call Stack





Callstack

```
public static int doMath(int a, int b){
```

```
    int mathResult;
```

```
    int halfOfA = divide(a, 2) // 3
```

```
    mathResult = mult(halfOfA, b)
```

```
    return mathResult
```

```
}
```

We have
returned
from divide



doMath(6, 8)

\$ main()

Programs Call Stack





Callstack

```
public static int doMath(int a, int b){  
    int mathResult;
```

```
    int halfOfA = divide(a, 2); // 3
```

Call another
method



```
    mathResult = mult(halfOfA, b); //mult(3, 8)
```

```
    return mathResult
```

```
}
```

doMath(6, 8)

\$ main()

Programs Call Stack





Callstack



```
public static int mult(int num, int times){  
    int multResult = num * times  
  
    return multResult  
  
}
```

Put mult() on
the stack



mult(3, 8)

doMath(5, 6)

\$ main()

Programs Call Stack





Callstack

```
public static int mult(int num, int times){  
    int multResult = num * times  
  
    return multResult  
  
}
```



mult(3, 8)

doMath(5, 6)

\$ main()

Programs Call Stack



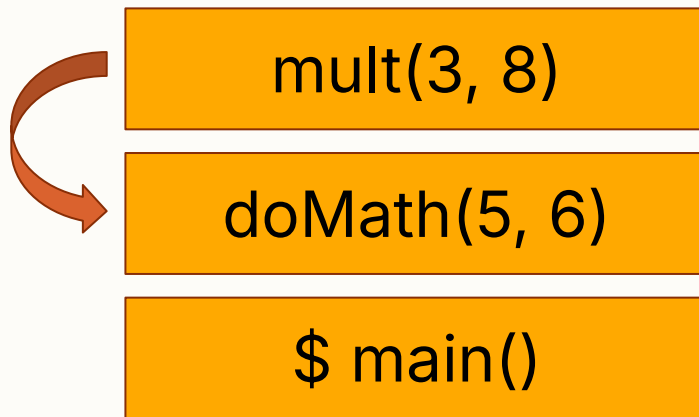


Callstack

```
public static int mult(int num, int times){  
    int multResult = num * times
```

```
    return multResult
```

```
}
```



Programs Call Stack





Callstack

```
public static int doMath(int a, int b){  
    int mathResult;
```

```
    int halfOfA = divide(a, 2); // 3
```

Returned
from mult()



```
    mathResult = mult(halfOfA, b); // 24
```

```
    return mathResult;
```

```
}
```

doMath(6, 8)

\$ main()

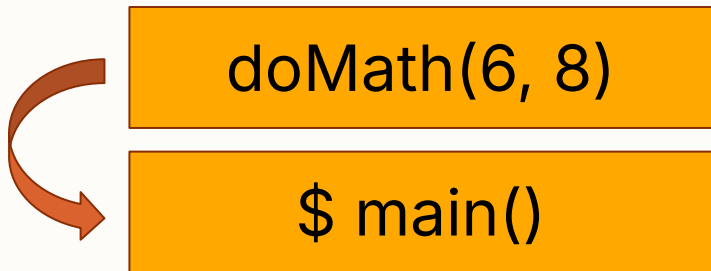
Programs Call Stack





Callstack

```
public static int doMath(int a, int b){  
    int mathResult;  
  
    int halfOfA = divide(a, 2); // 3  
  
    mathResult = mult(halfOfA, b); // 24  
  
    return mathResult;  
}
```



Programs Call Stack






Callstack

```
public static void main(String[] args){
```

```
    int a = 6;
```

```
    int b = 8;
```

Returned from
doMath()

 `int result = doMath(a, b); // 24`

```
    System.out.print(result);
```

```
}
```

\$ main()

Programs Call Stack





Callstack

```
public static void main(String[] args){  
    int a = 6;  
    int b = 8;  
  
    int result = doMath(a, b); // 24  
  
    System.out.print(result); // 24  
  
}
```



\$ main()

Programs Call Stack





Callstack

```
public static void main(String[] args){
```

```
    int a = 6;
```

```
    int b = 8;
```

```
    int result = doMath(a, b); // 24
```

```
    System.out.print(result); // 24
```

```
    return; // omitted
```

```
}
```



\$ main()

Programs Call Stack





Callstack

```
public static void main(String[] args){
```

```
    int a = 6;
```

```
    int b = 8;
```

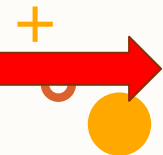
```
    int result = doMath(a, b); // 24
```

```
    System.out.print(result); // 24
```

```
    return; // omitted
```

```
}
```

Program finished
when callstack is
empty



Programs Call Stack



Callstack

- You can see how this would scale for larger programs
 - Large call stacks
 - Even for methods you don't write





Callstack

- When we called a method:
 - All of its parameters got “filled in” as variables we can use locally

- Let's keep this idea in mind going forward

mult(3, 8)

doMath(5, 6)

\$ main()





Callstack

- Consider the following method
- What would our callstack look like?

```
public static void doThing(){  
    // do anything  
  
    doThing()  
}
```

\$ main()


Programs Call Stack





Callstack

- Consider the following method
- What would our callstack look like?



```
public static void doThing(){  
    // do anything  
  
    doThing()  
}
```

doThing()

\$ main()

Programs Call Stack





Callstack

- Consider the following method
- What would our callstack look like?

```
public static void doThing(){  
    // do anything
```



doThing()

}

doThing()

\$ main()

Programs Call Stack





Callstack

- Consider the following method
- What would our callstack look like?



```
public static void doThing(){  
    // do anything
```

```
    doThing()  
}
```

doThing()

doThing()

\$ main()

Programs Call Stack





Callstack

- Consider the following method
- What would our callstack look like?

```
public static void doThing(){  
    // do anything
```

```
doThing()  
}
```

doThing()

doThing()

\$ main()

Programs Call Stack





Callstack

- Consider the following method
- What would our call



```
public static void doThing(){  
    // do anything  
  
    doThing()  
}
```

doThing()

doThing()

doThing()

\$ main()

Programs Call Stack





Callstack

- Consider the following
- What would our call



```
public static void doThing(){  
    // do anything  
  
    doThing()  
}
```

doThing()

doThing()

doThing()

doThing()

\$ main()

Programs Call Stack





Callstack

- Consider the following
- What would our call



```
public static void doThing(){  
    // do anything  
  
    doThing()  
}
```

doThing()

doThing()

doThing()

doThing()

doThing()

\$ main()


Programs Call Stack





Callstack

- Consider the following
- What would our call



```
public static void doThing(){  
    // do anything  
  
    doThing()  
}
```

doThing()

doThing()

doThing()

doThing()

doThing()

doThing()

\$ main()

Programs Call Stack





Callstack

- StackOverflowError
 - Let's see this
- Callstack has a "height" limit
- Cannot exceed it
- We need a way to stop it from running after a point





Callstack

- Let's look at this method instead

➔ `public static void doThing(int timesRan){
 if (timesRan >= 3) {
 return
 }`

`doThing(timesRan + 1)`

`}`

doThing(0)

\$ main()

Programs Call Stack





Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
}
```

 doThing(timesRan + 1)

 return
 }

doThing(0)

\$ main()

Programs Call Stack



Callstack

- Let's look at this method instead

➔ public static void doThing(int timesRan){
 if (timesRan >= 3) {
 return
 }

doThing(timesRan + 1)

+
○ }
●
return

doThing(1)

doThing(0)

\$ main()

Programs Call Stack



Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
}
```

 doThing(timesRan + 1)

+
○ }


doThing(1)

doThing(0)

\$ main()

Programs Call Stack



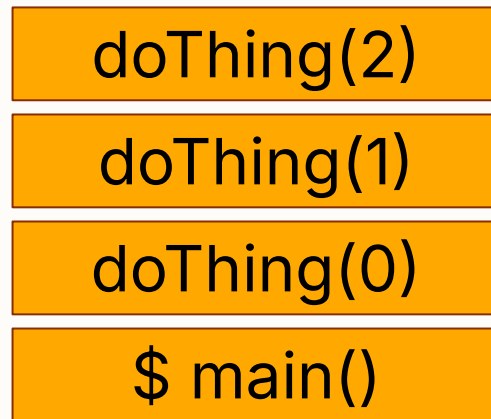
Callstack

- Let's look at this method instead

➔ public static void doThing(int timesRan){
 if (timesRan >= 3) {
 return
 }

 doThing(timesRan + 1)

 return
}



Programs Call Stack



Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
    doThing(timesRan + 1)  
    return  
}
```

 doThing(timesRan + 1)

+
○ }


doThing(2)

doThing(1)

doThing(0)

\$ main()

Programs Call Stack



Callstack

- Let's look at this method instead

→ public static void doThing(int timesRan) {
 if (timesRan >= 3) {
 return
 }

 doThing(timesRan + 1)

 return
}

doThing(3)

doThing(2)

doThing(1)

doThing(0)

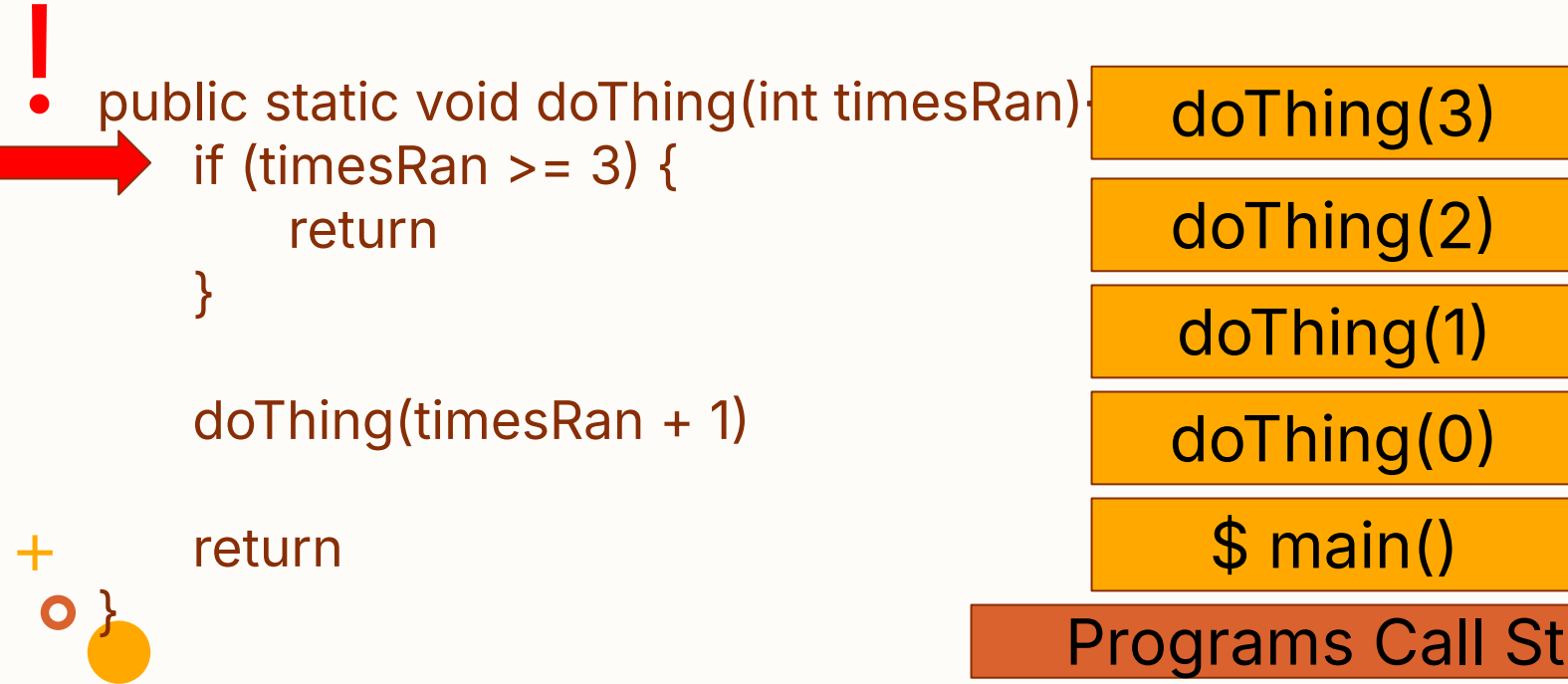
\$ main()

Programs Call Stack



Callstack

- Let's look at this method instead



Programs Call Stack



Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan) {  
    if (timesRan >= 3) {  
        return  
    }  
    doThing(timesRan + 1)  
}
```



doThing(3)

doThing(2)

doThing(1)

doThing(0)

\$ main()

Programs Call Stack

doThing(timesRan + 1)

return





Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
    doThing(timesRan + 1)  
    return  
}
```



doThing(timesRan + 1)



return

doThing(2)

doThing(1)

doThing(0)

\$ main()

Programs Call Stack



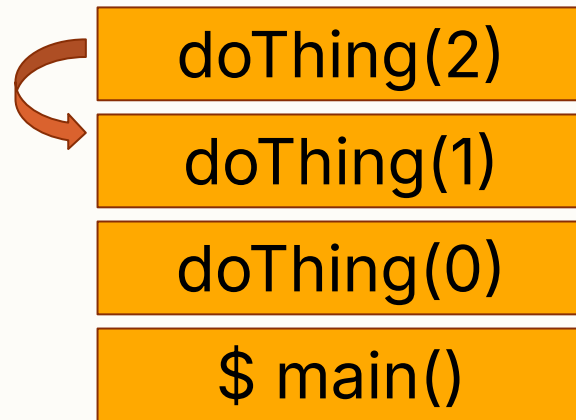
Callstack

- Let's look at this method instead

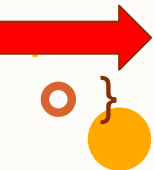
```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
    doThing(timesRan + 1)  
}
```

doThing(timesRan + 1)

return



Programs Call Stack





Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
    doThing(timesRan + 1)  
    return  
}
```



doThing(timesRan + 1)



return

doThing(1)

doThing(0)

\$ main()

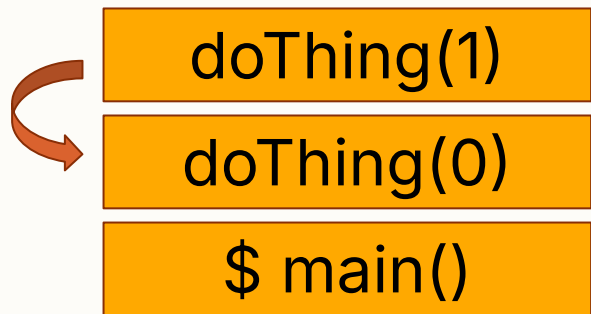
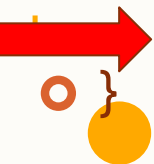
Programs Call Stack



Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
    doThing(timesRan + 1)  
    return  
}
```



Programs Call Stack



Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
}
```

 doThing(timesRan + 1)

+
○ }


return

doThing(0)

\$ main()

Programs Call Stack



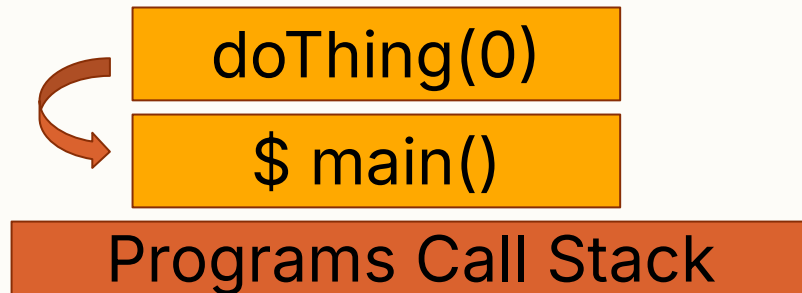
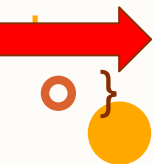
Callstack

- Let's look at this method instead

```
public static void doThing(int timesRan){  
    if (timesRan >= 3) {  
        return  
    }  
}
```

doThing(timesRan + 1)

return





Callstack

- We were able to write a method that
 - Calls itself within its body
 - Any number of times
 - But eventually terminated
- Made its way back down the callstack





Recursion

- This idea is called recursion
 - Anytime a function is calling itself
- Recursion functions according to the exact same rules any other method does
- Not "special" in that regard





Recursion

- Recursive functions *usually* can be broken down into two primary components
- Base Case
- Simplifying the problem





Recursion

- Base Case:
 - Solve the simplest version of the problem that you can
 - Usually takes form of *if param == small case*
 - If this isn't the case, simplify





Recursion

- Simplifying the problem
 - If your input is not in its smallest ideal form
 - Try to simplify your problem, then call the method again
 - Recombine elements to build solution





Callstack

- We can use this idea to solve real problems
- Those two components can be unintuitive until you see them
- Let's try to solve that `countOccurrences()` problem again





Callstack

- Base case
 - Simplest form of problem
 - Empty String has 0 occurrences
- Look at first character
 - If first character is target, add one
 - Else continue





Callstack

- Look at first character
 - If first character is target, add one
 - Else continue

H	E	L	L	O
---	---	---	---	---





H	E	L	L	O
---	---	---	---	---

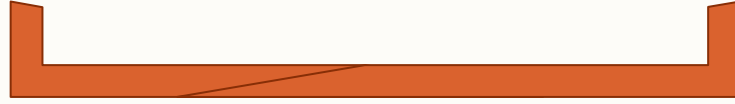


0 + countOccurrences(target, "ello")

Target = L



H	E	L	L	O
---	---	---	---	---



0 +

E	L	L	O
---	---	---	---



0 +

1 + countOccurrences(target, "llo")

Target = L



H	E	L	L	O
---	---	---	---	---



0 +

E	L	L	O
---	---	---	---



0 +

0 +

L	L	O
---	---	---

Target = L



0 +

0 +

1 + countOccurrences(target, "lo")



0 +

0 +

1 +

L	O
---	---



1 + countOccurrences(target, "o")

Target = L



0 +

0 +

1 +

L	O
---	---



1 + countOccurrences(target, "o")

0 +

0 +

1 +

1 +

O



0 + countOccurrences(target, "")

Target = L



0 +

0 +

1 +



1 + countOccurrences(target, "o")

0 +

0 +

1 +

1 +



0 + countOccurrences(target, "")

Target = L

0 +

0 +

1 +

1 +

0 +

0



= 2





Callstack

- Let's do another simple recursion problem
- Discrete Math
 - Fibonacci Sequence
 - Compute the Nth fibonacci number





Callstack

- Let's try countOccurrences again
 - Without loops
 - Common patterns





Callstack

- Let's try countOccurrences again
 - Without loops
 - Common patterns
- Was difficult to accomplish without having another piece of state
 - Something that persists across methods call





Callstack

- Common Recursion pattern:
- Helper methods:
- User friendly `countOccurrences()`
 - Less friendly, hidden `countOccurrencesHelper()`





More recursion

- All problems that could be solved with loops:
 - Can be solved via recursion
- Some problems fit themselves nicely to recursion
 - Fib(n)
 - Factorial(n), etc.





More recursion

- We're gonna do 7ish recursion problems
 - ~50% from CodingBat
- Common interview questions
- Start easier ones
- Get harder





More recursion

- EndX
 - Input \Rightarrow string
 - Return \Rightarrow string with all 'x's moved to the end
- "Hexllxo" \Rightarrow "Helloxx"
- "xx_java_dev__xx" \Rightarrow "_java_dev_xxxx"





More recursion

- Print Triangle
- Input \Rightarrow Height
- Return \Rightarrow None
 - Should print a triangle of size *height* to the screen





More recursion

- Is Palindrome
- Input \Rightarrow String to check
- Return \Rightarrow T/F if string is a palindrome

- "hello" \Rightarrow false
- "racecar" \Rightarrow true





More recursion

- Count 11's in an array
- Input \Rightarrow input array
- Return \Rightarrow how many 11's were present in that array

- $[4, 5, 6, 11, 5] \Rightarrow 1$
- $[11, 45, 12, 10, 11] \Rightarrow 2$





More recursion

- Insert * into between duplicate letters
 - Input \Rightarrow String to insert
 - Return \Rightarrow String with * between any duplicate letter
-
- "hello" \Rightarrow "hel*lo"
 - "aabb" \Rightarrow "a*ab*b"
 - "aaa" \Rightarrow "a*a*a"





More recursion

- Insert a value into a linked list
 - Input \Rightarrow value and index
 - Return \Rightarrow None, modify list
-
- We'll use this weeks LP code to start with





More recursion

- Find index of value
- Input \Rightarrow sorted array, target value
- Return \Rightarrow index of target value

- Special algorithm
- Binary search
- Very fast $\Rightarrow O(n \log n)$ time

