Callstacks and Recursion

Callstack

- We've briefly talked about the callstack
  - Stack ADT
  - Method call ⇒ Stack.push()
  - Return ⇒ 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)

}
```

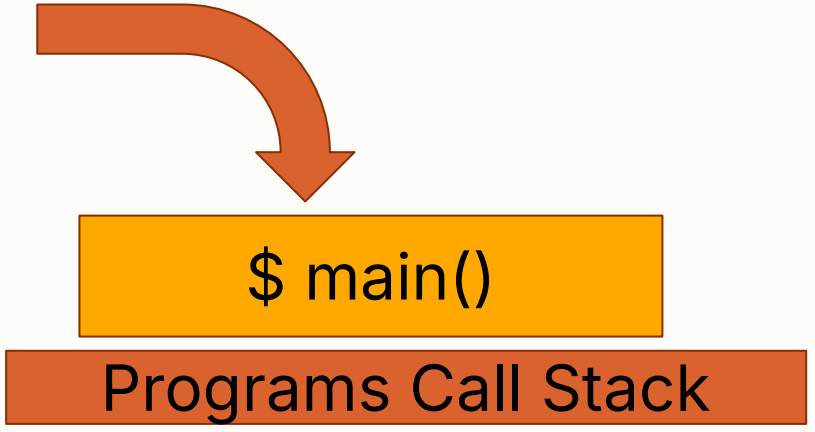
Programs Call Stack

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



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)

}
```

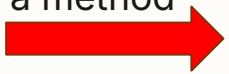
$ main()

Programs Call Stack

Callstack

```
public static void main(String[] args){
    int a = 6;
    int b = 8

    int result = doMath(a, b)

    System.out.print(result)

}
```

Need to call a method



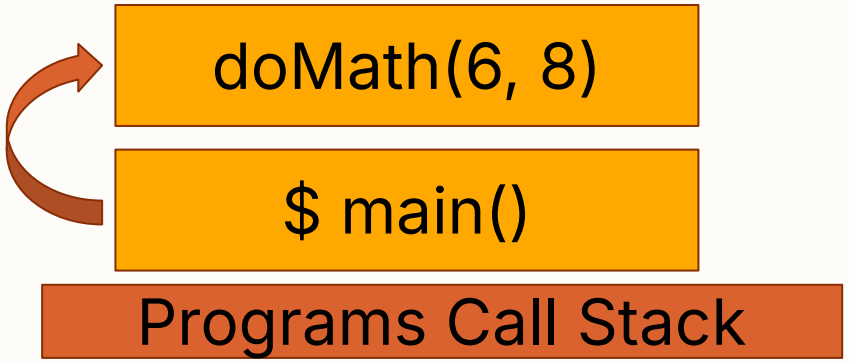
$ main()

Programs Call Stack

Callstack

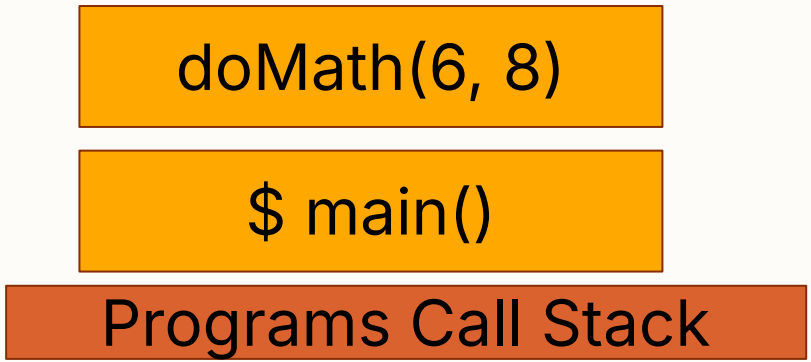
```
public static int doMath(int a, int b){
    int mathResult;

    int halfOfA = divide(a, 2)

    mathResult = mult(halfOfA, b)

    return mathResult

}
```

Put doMath()
on the stack



Callstack

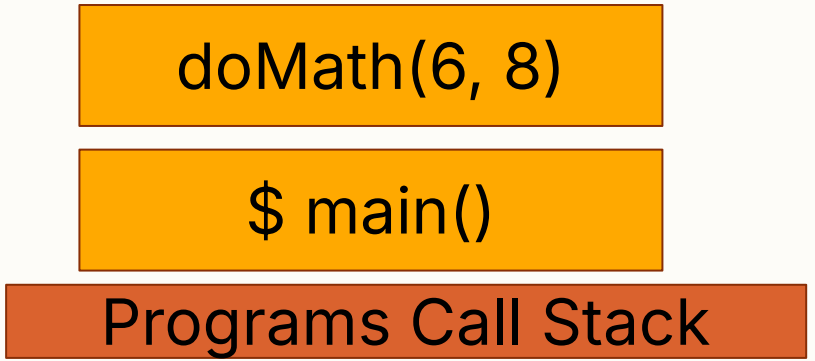
```
public static int doMath(int a, int b){
    int mathResult;

    int halfOfA = divide(a, 2)

    mathResult = mult(halfOfA, b)

    return mathResult

}
```



Callstack

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

    int halfOfA = divide(a, 2)

    mathResult = mult(halfOfA, b)

    return mathResult

}
```

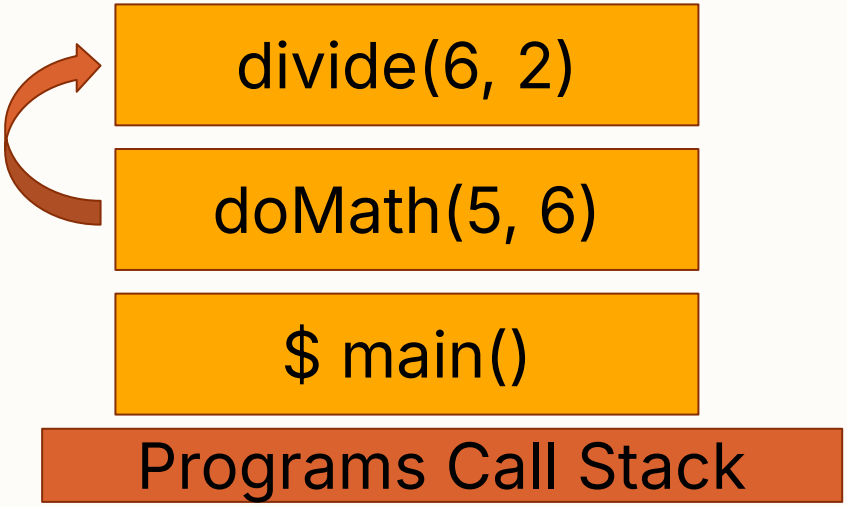
Need to call a method



Callstack

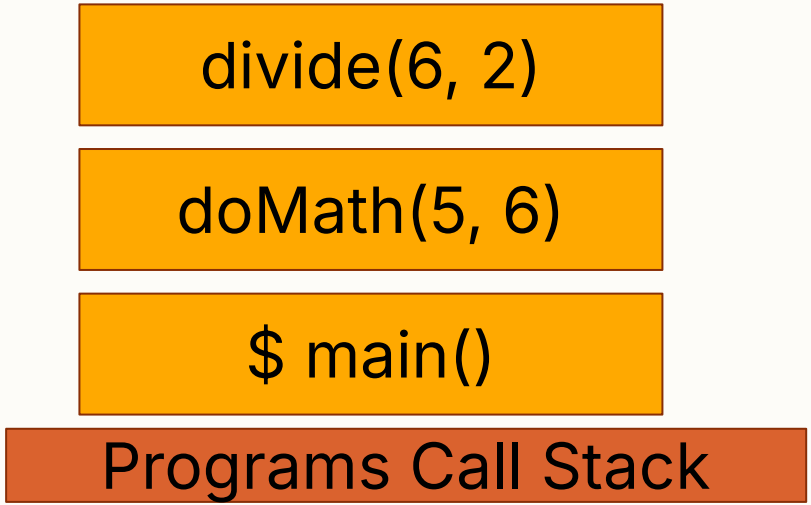
```
public static int divide(int n, int d){
        int result = n / d;

        return result;
}
```

Put divide() on the stack



Callstack

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

    return result;
}
```

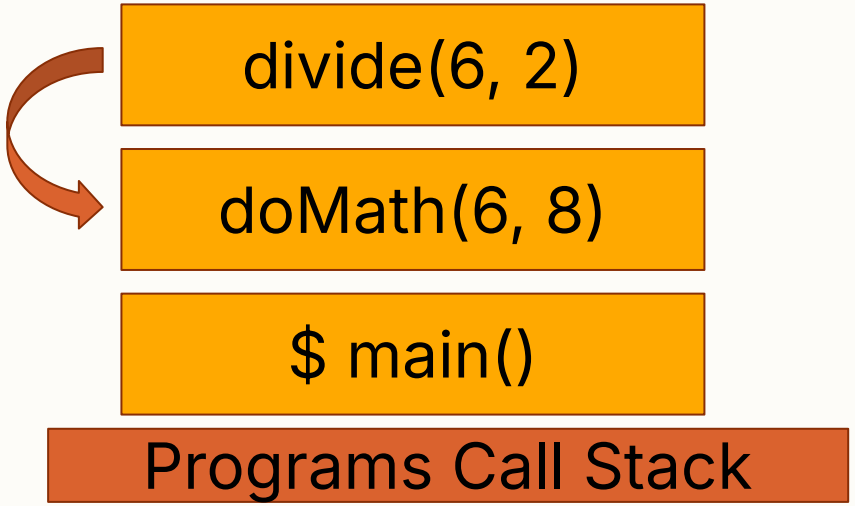


Callstack

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

    return result;
}
```

Return ⇒
stack pop

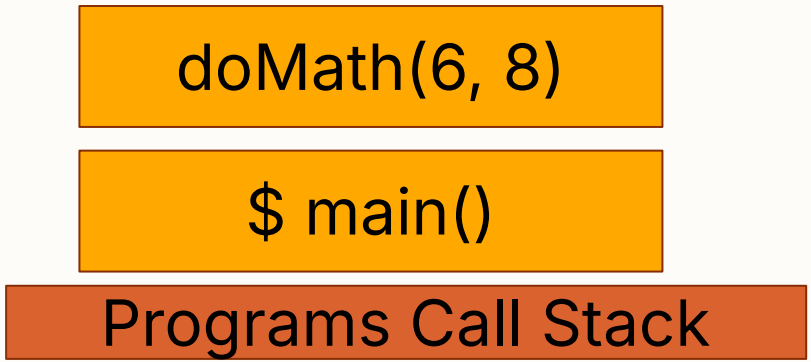
Return right back to where we left off in doMath



Callstack

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

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

    mathResult = mult(halfOfA, b)

    return mathResult

}
```

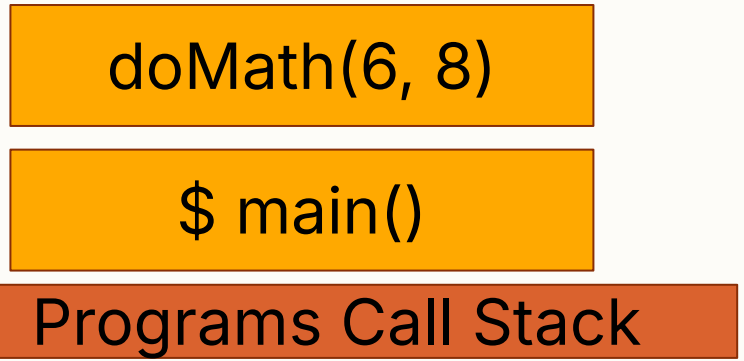
We have returned from divide



Callstack

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

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

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

    return mathResult

}
```

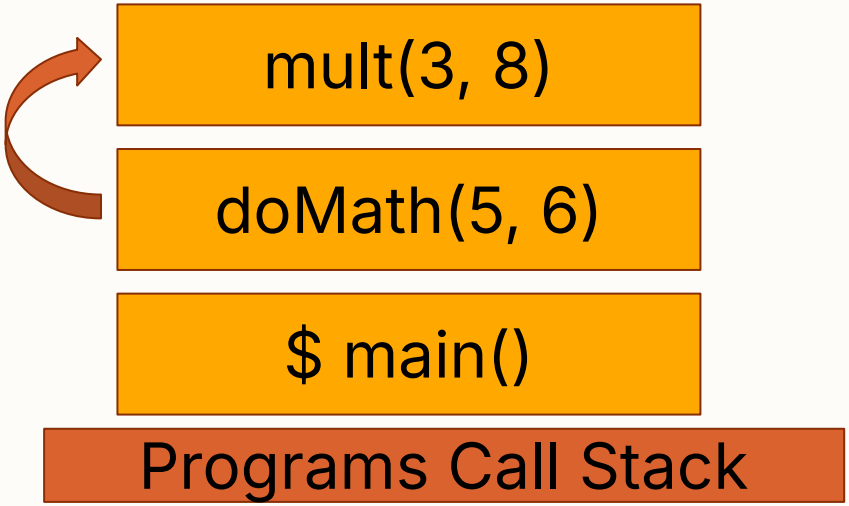
Call another method



Callstack

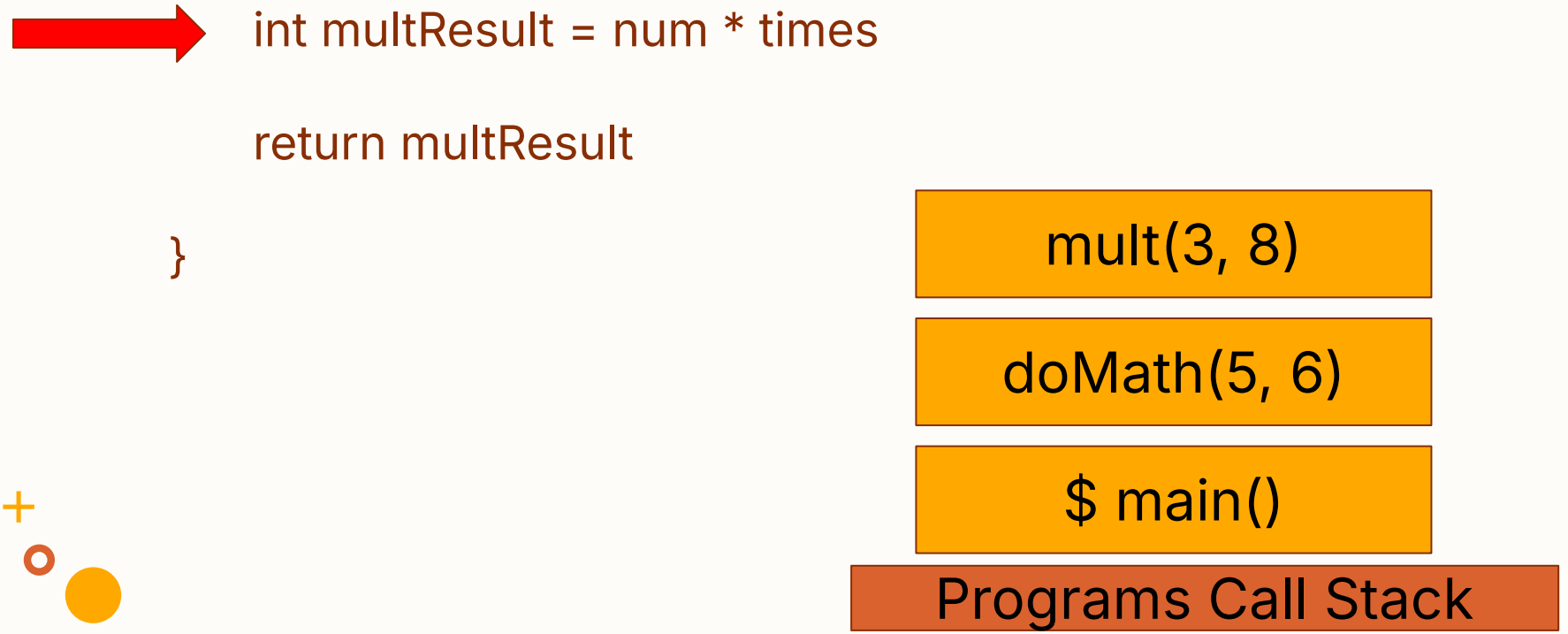
```
public static int mult(int num, int times){
    int multResult = num * times

    return multResult

}
```

Put mult() on the stack



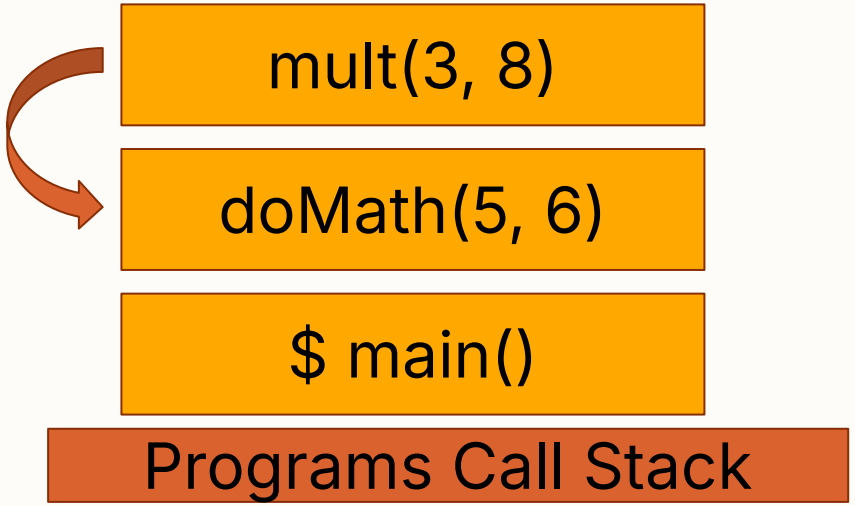
Callstack

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

    return multResult

}
```



Callstack

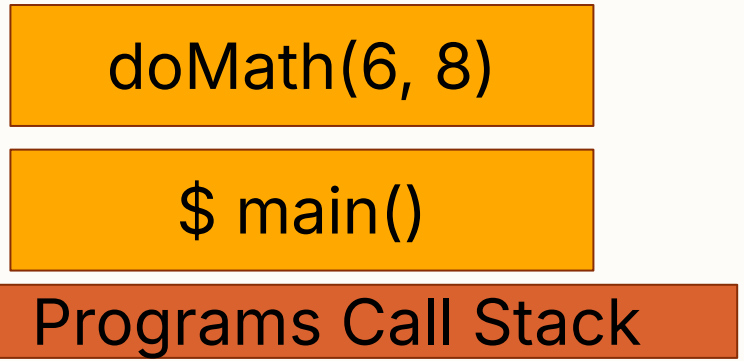
```
public static int mult(int num, int times){
    int multResult = num * times

    return multResult

}
```



Callstack

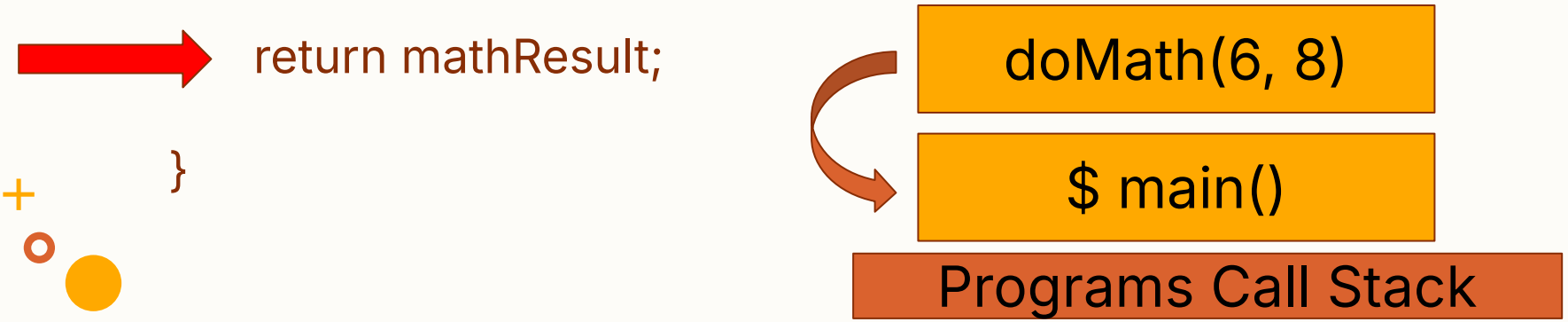
```
public static int doMath(int a, int b){
    int mathResult;

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

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

    return mathResult;

}
```

Returned from mult()



Callstack

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

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

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

    return mathResult;

}
```



Callstack

```
public static void main(String[] args){
        int a = 6;
        int b = 8;

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

        System.out.print(result);

}
```

Returned from doMath()

$ main()

Programs Call Stack

Callstack

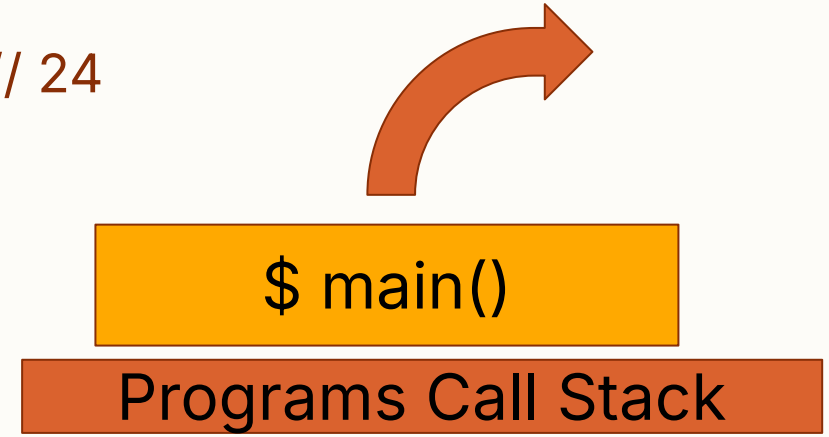
```
public static void main(String[] args){
        int a = 6;
        int b = 8;

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

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

}
```

$ main()

Programs Call Stack

Callstack

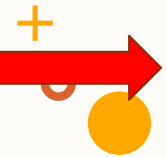
```
public static void main(String[] args){
        int a = 6;
        int b = 8;

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

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

        return; // omitted

}
```



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

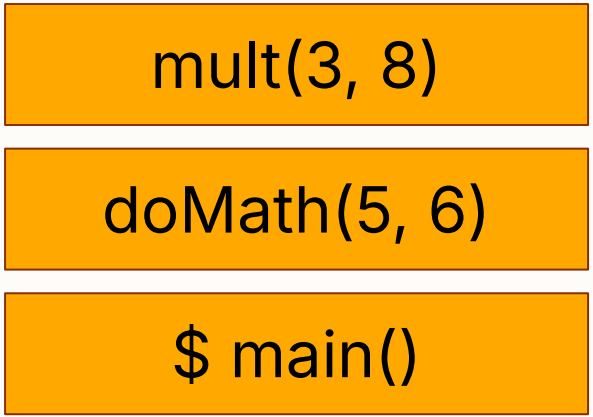
Callstack

- 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

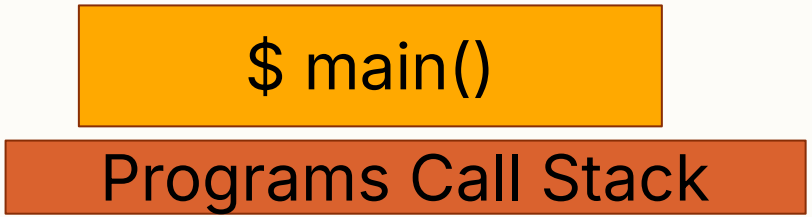
- Let’s keep this idea in mind going forward



Callstack

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

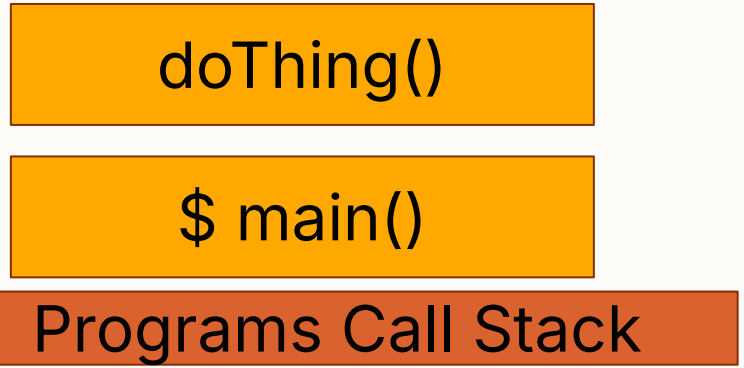
```
public static void doThing(){
    // do anything

    doThing()
}
```



Callstack

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

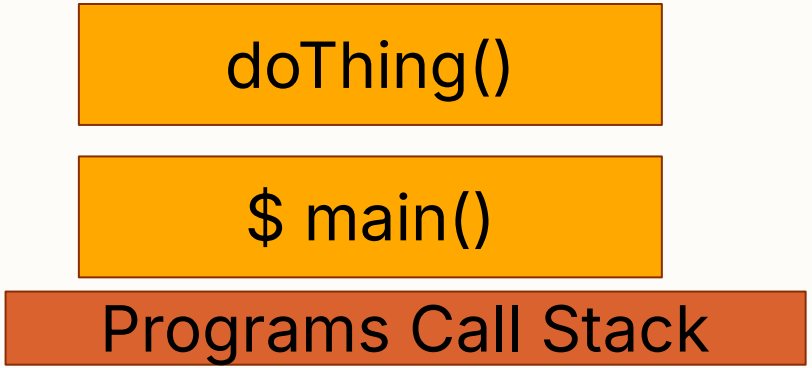
```
public static void doThing(){
    // do anything

    doThing()
}
```



Callstack

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

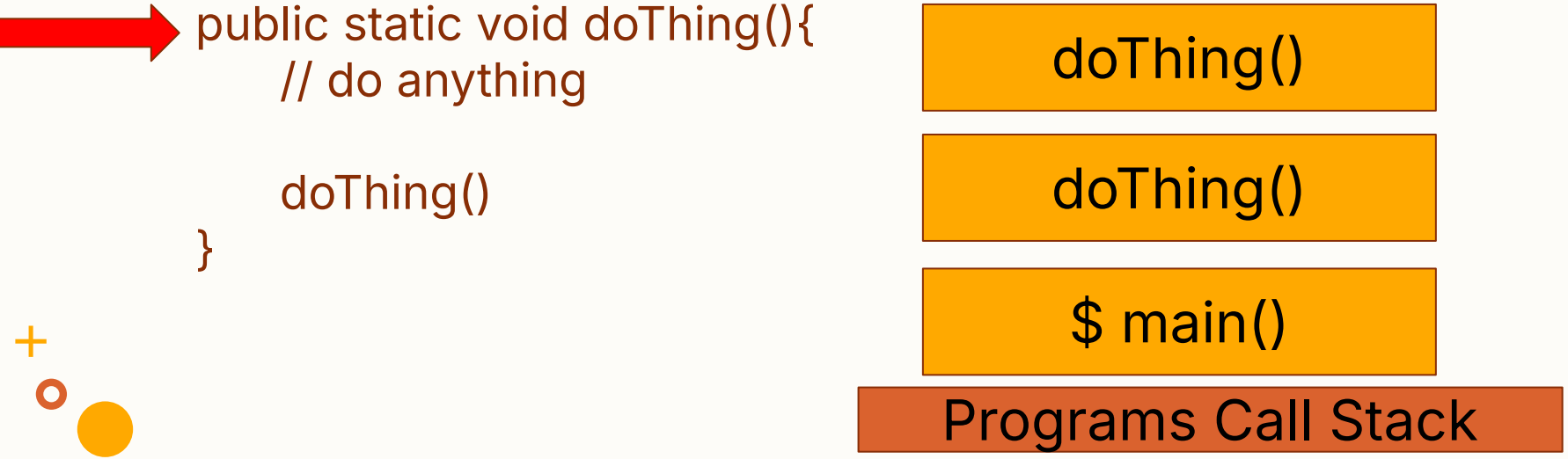
```
public static void doThing(){
    // do anything

    doThing()
}
```



Callstack

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

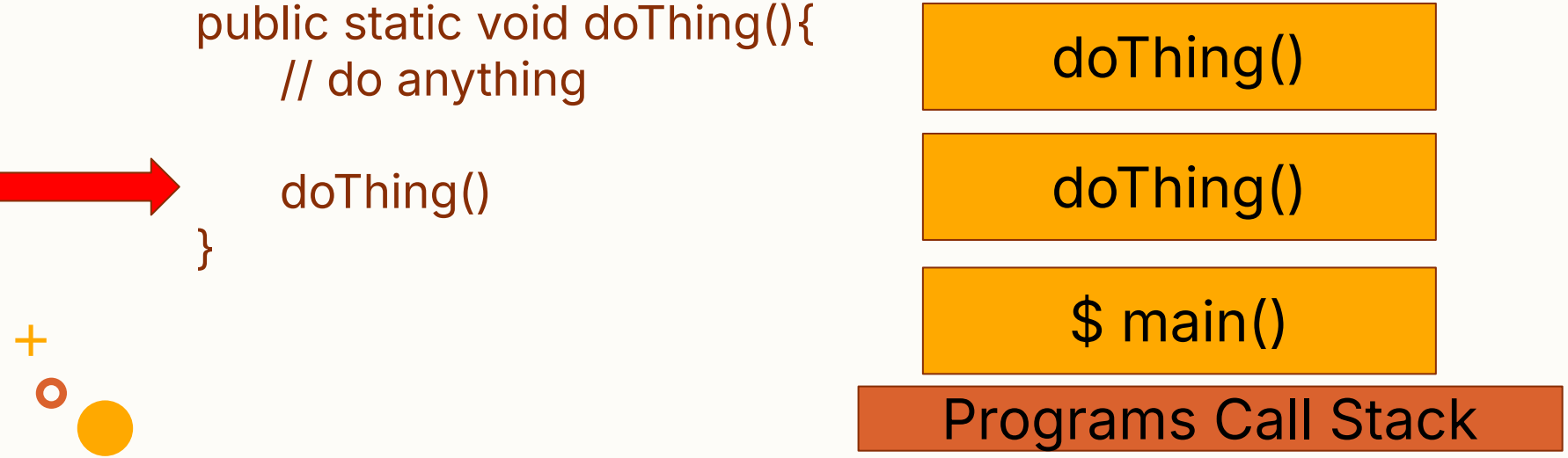
```
public static void doThing(){
    // do anything

    doThing()
}
```



Callstack

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

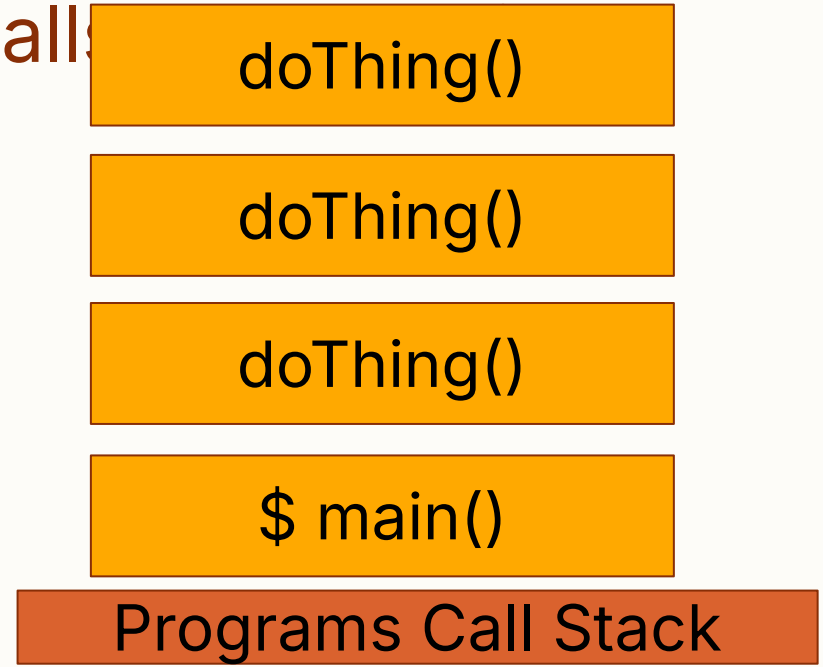
```
public static void doThing(){
    // do anything

    doThing()
}
```



Callstack

- Consider the following method
- What would our call

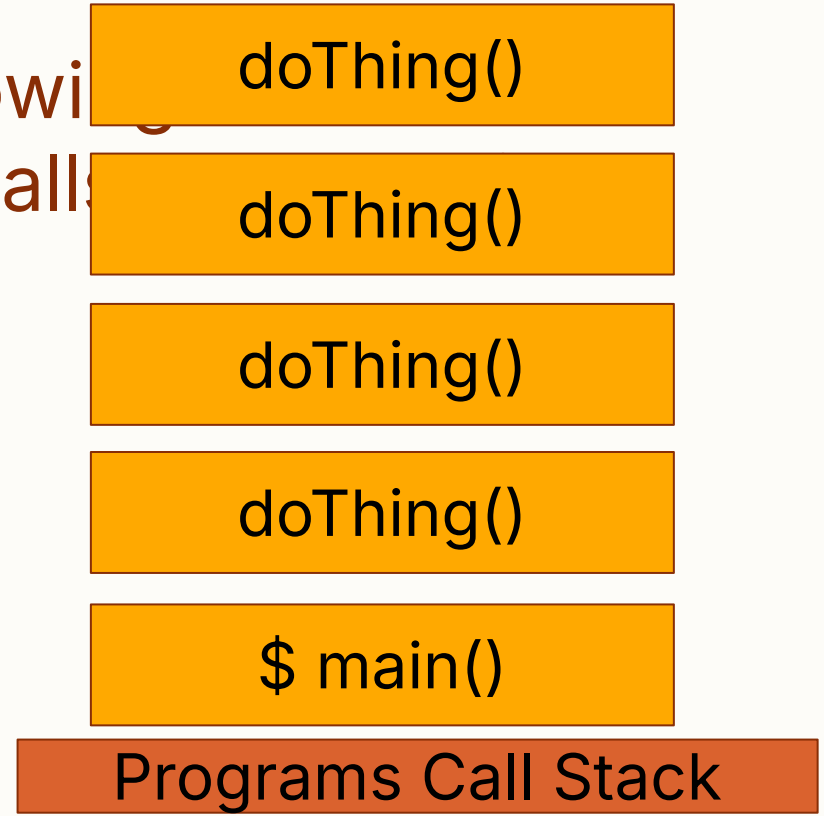
```
public static void doThing(){
    // do anything

    doThing()
}
```



Callstack

- Consider the followi
- What would our call

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

    doThing()
}
```



Callstack

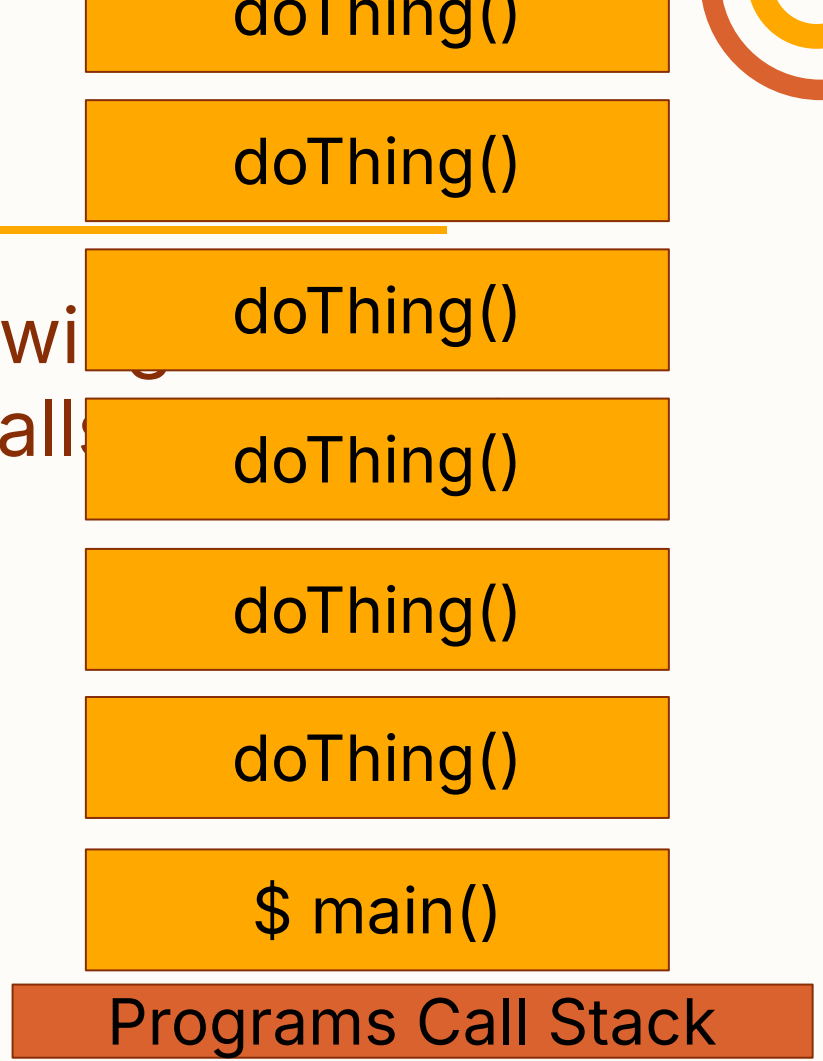
- Consider the followi
- What would our call

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

    doThing()
}
```

Callstack

- Consider the followi
- What would our call

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

    doThing()
}
```



Callstack

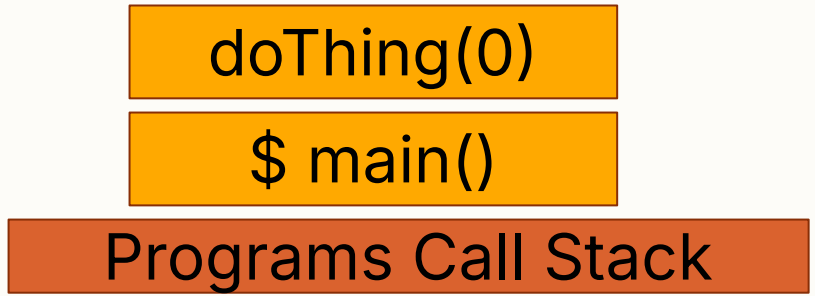
- StackOverFlowError
  - Let’s see this

- 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

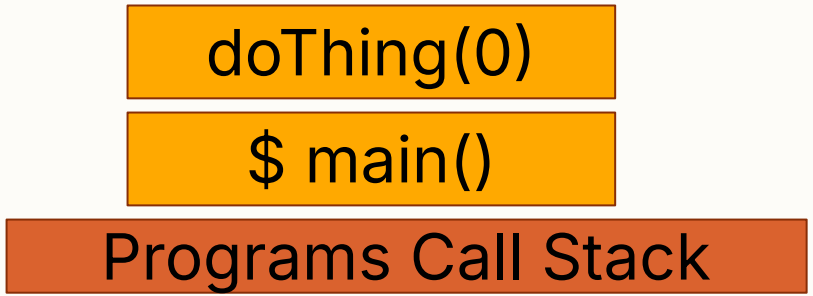
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)
}
```



Callstack

- Let’s look at this method instead

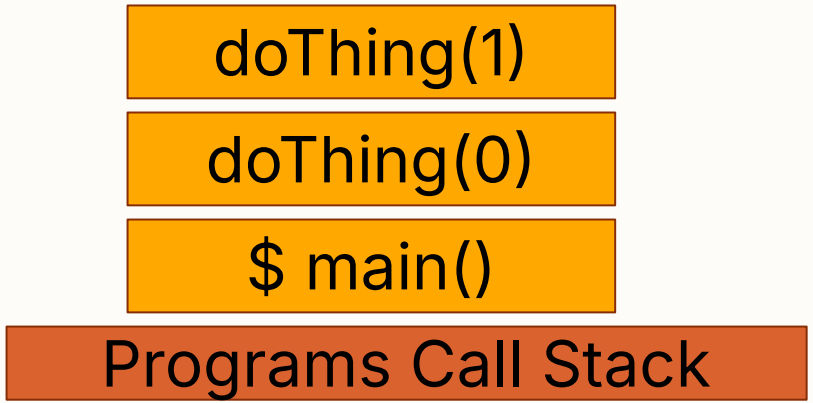
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let’s look at this method instead

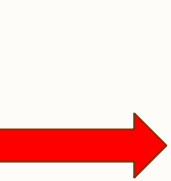
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```

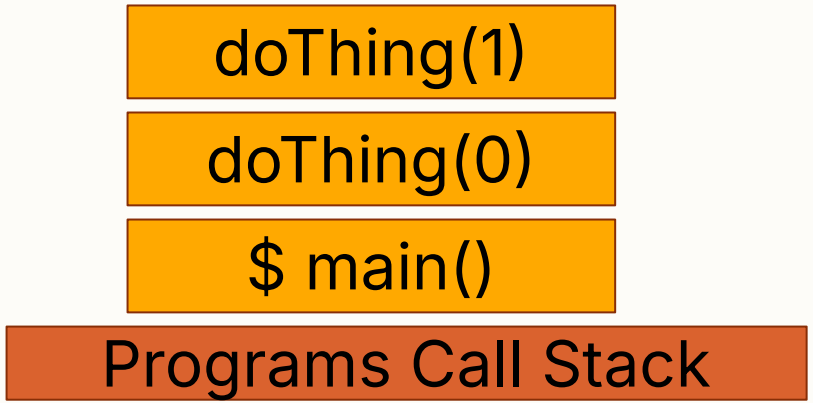


Callstack

- Let’s look at this method instead

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

    doThing(timesRan + 1)

    return
}
```

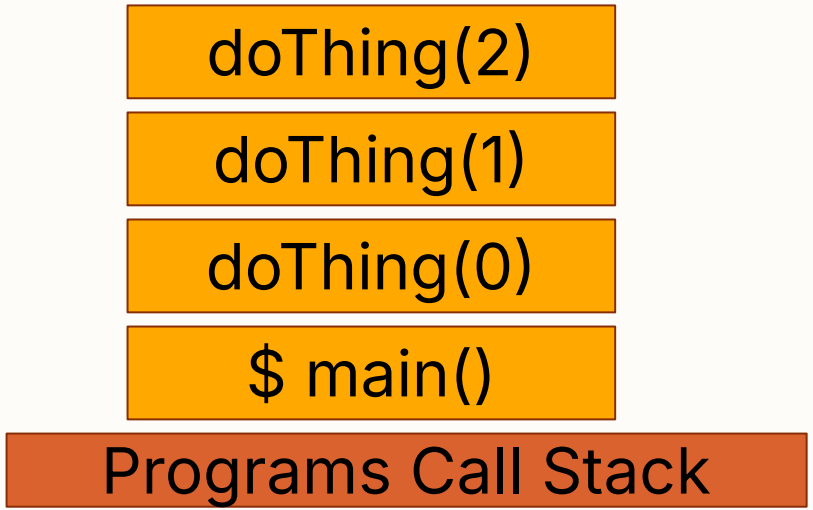




Callstack

- Let's look at this method instead

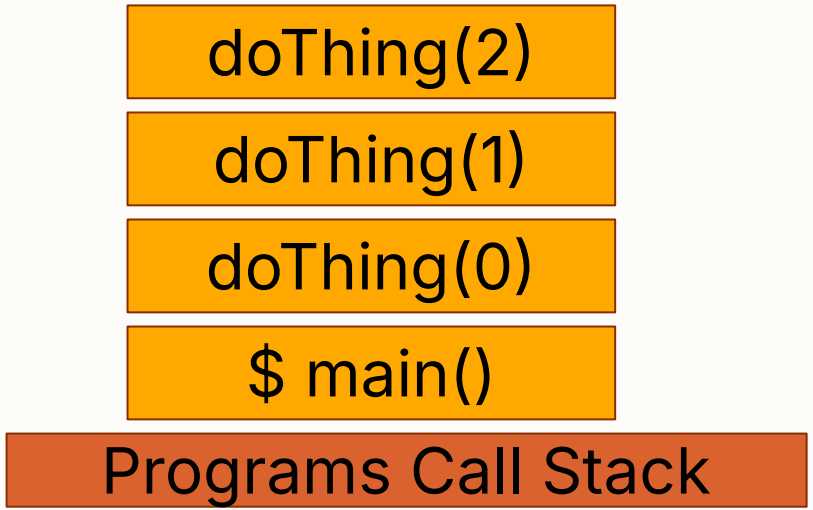
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let’s look at this method instead

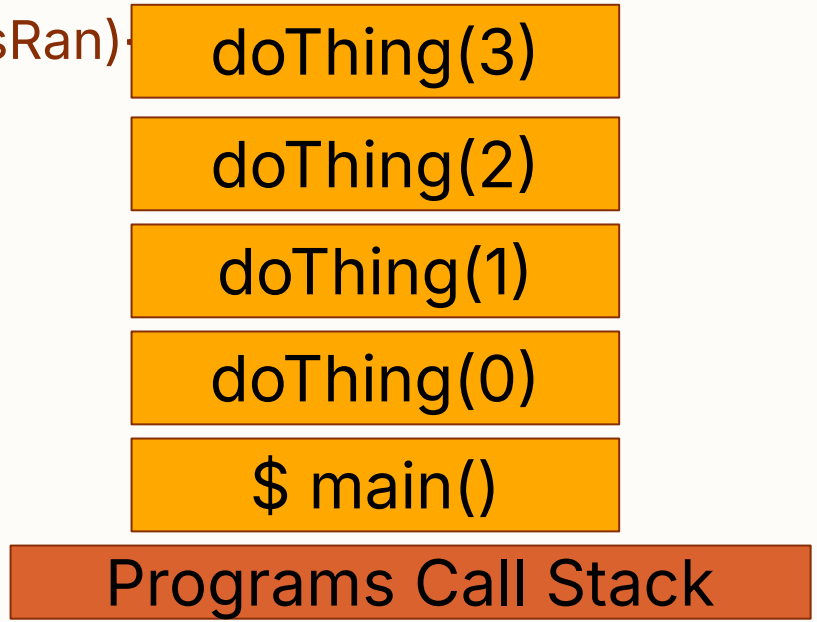
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let's look at this method instead

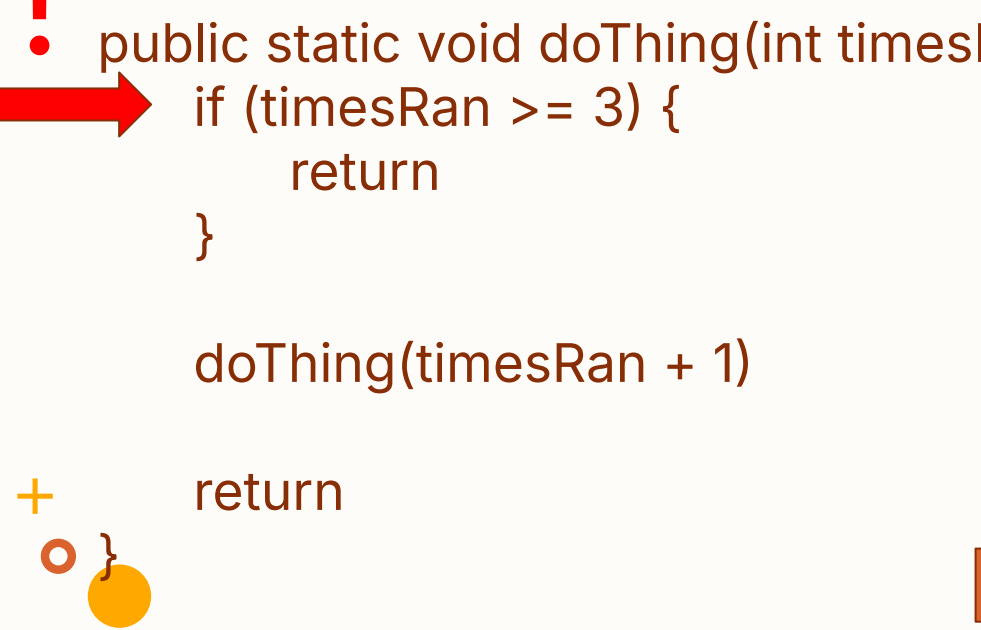
```
public static void doThing(int timesRan) {
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let's look at this method instead

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

    doThing(timesRan + 1)

    return
}
```



Callstack

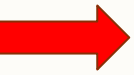
- Let's look at this method instead

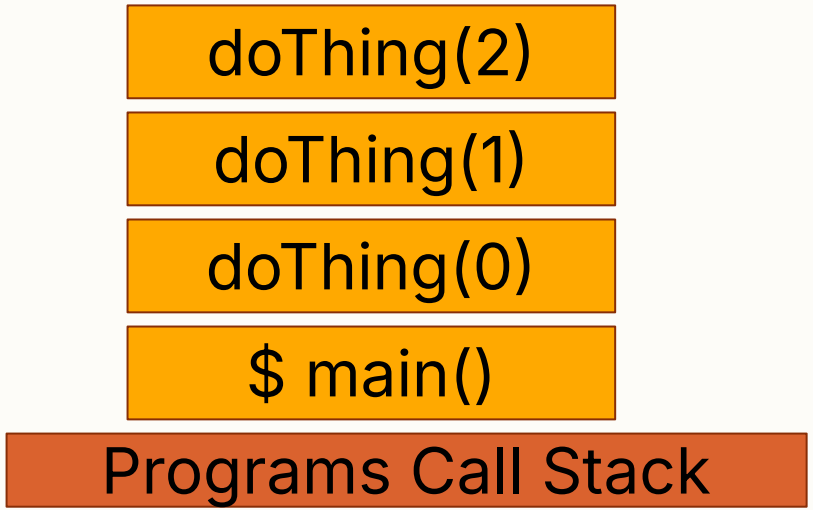
```
public static void doThing(int timesRan) {
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```

Callstack

- Let’s look at this method instead

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

    doThing(timesRan + 1)

    return
}
```

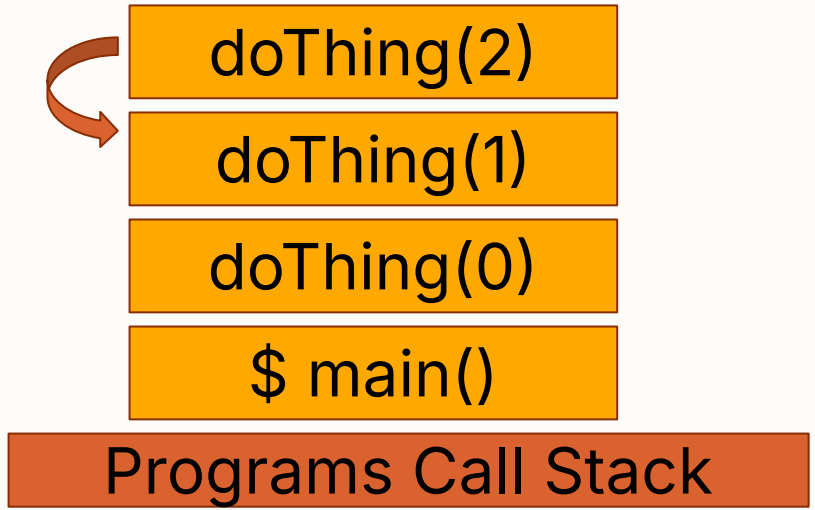




Callstack

- Let’s look at this method instead

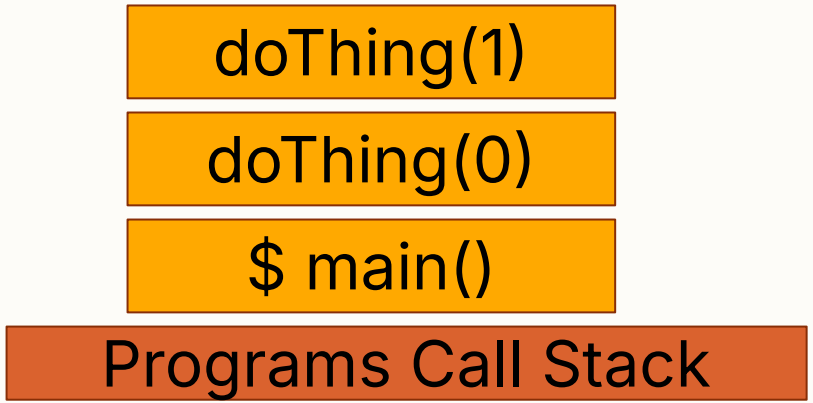
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let’s look at this method instead

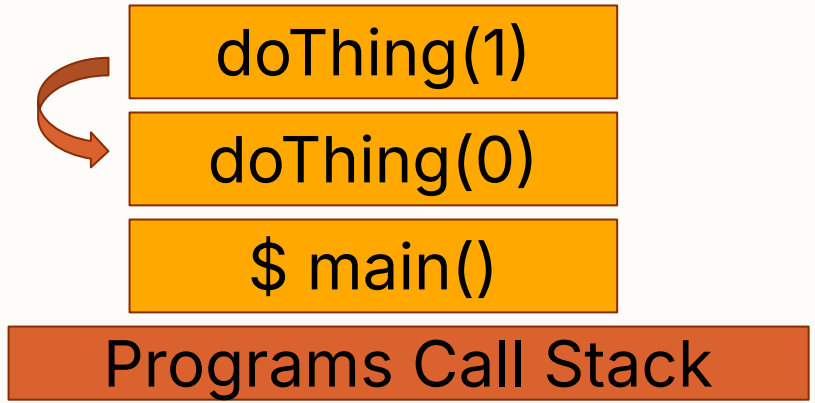
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let’s look at this method instead

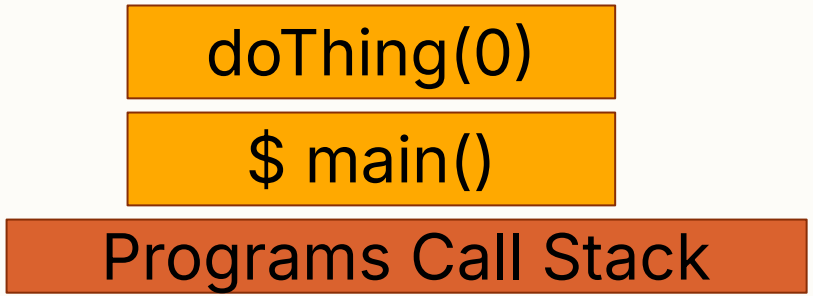
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    doThing(timesRan + 1)

    return
}
```



Callstack

- Let’s look at this method instead

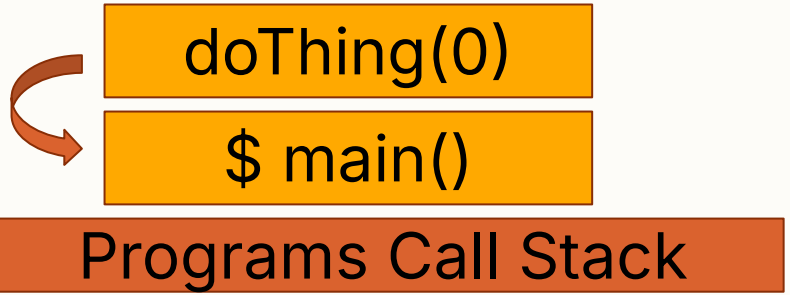
```
public static void doThing(int timesRan){
    if (timesRan >= 3) {
        return
    }

    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
}
```



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

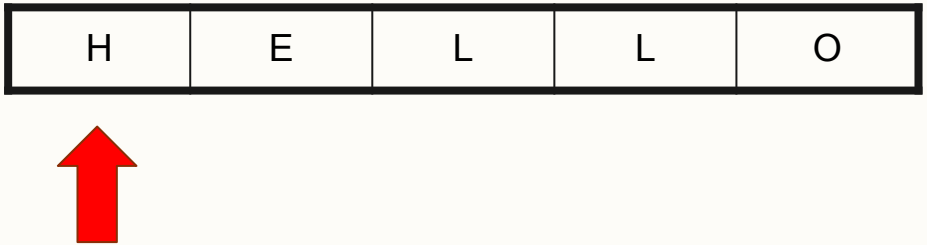
- Let’s try to solve that countOccurances() problem again

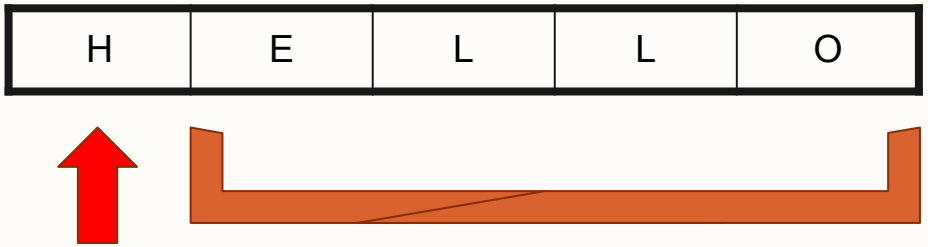
Callstack

- 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

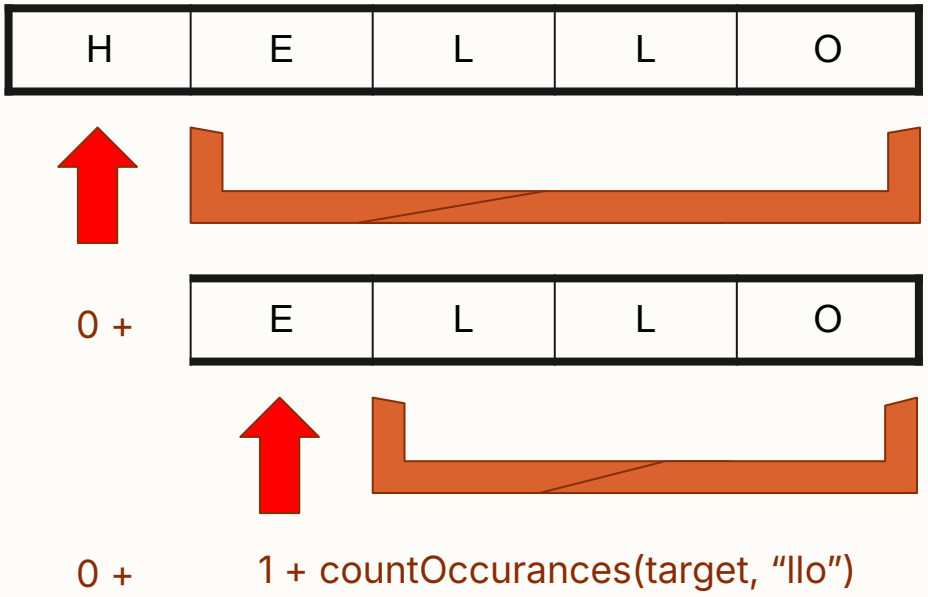
- Look at first character
  - If first character is target, add one
  - Else continue



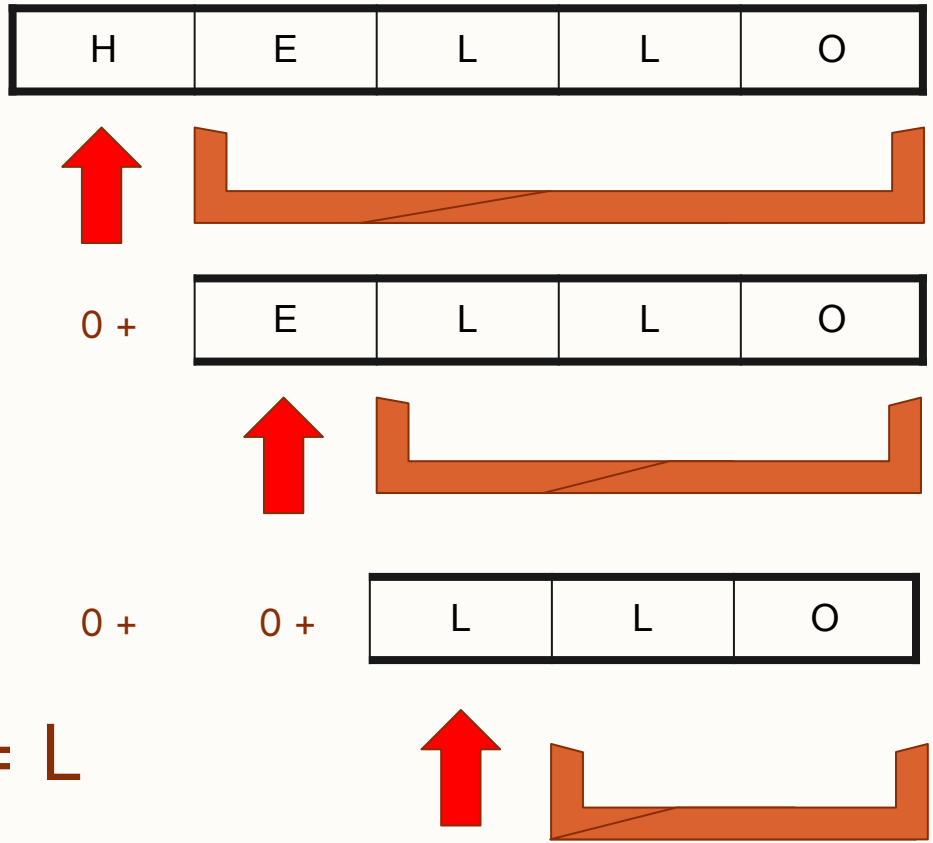


0 + countOccurances(target, “ello”)

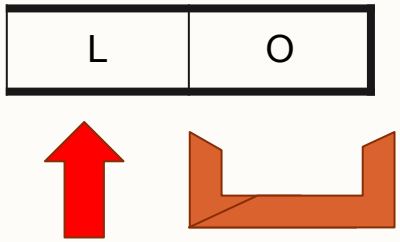
Target = L



Target = L


H E L L O
0 + E L L O
0 + 0 + L L O
Target = L
0 + 0 + 1 + countOccurances(target, “lo”)

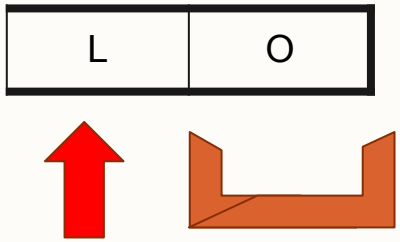
0 + 0 + 1 +



1 + countOccurances(target, “o”)

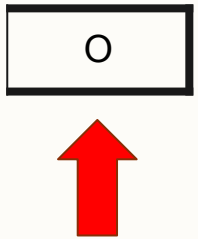
Target = L

0 + 0 + 1 +



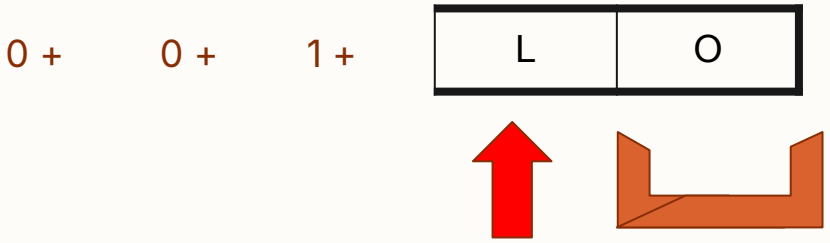
1 + countOccurances(target, “o”)

0 + 0 + 1 + 1 +

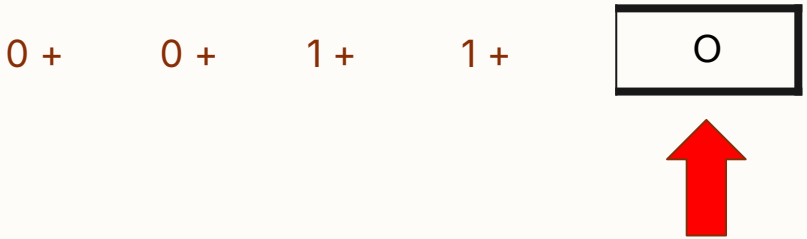


0 + countOccurances(target, “”)

Target = L

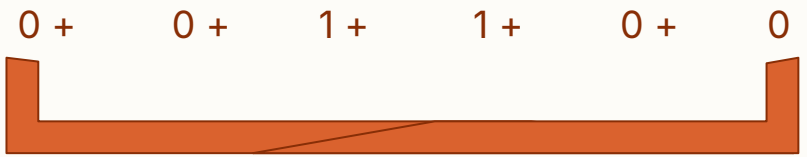


1 + countOccurances(target, “o”)



0 + countOccurances(target, “”)

Target = L



= 2

Callstack

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

Callstack

- Let’s try countOccurances again
  - Without loops
  - Common patterns

Callstack

- Let’s try countOccurances 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 countOccurances()
  - Less friendly, hidden countOccurancesHelper()

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 ⇒string
  - Return ⇒string with all ‘x’s moved to the end

- “Hexllxo” ⇒ “Helloxx”
- “xx_java_dev__xx” ⇒ “_java_dev_xxxx”

More recursion

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

More recursion

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

- “hello” ⇒ false
- “racecar” ⇒ true

More recursion

- Count 11’s in an array
- Input ⇒ input array
- Return ⇒ how many 11’s were present in that array

- [4, 5 ,6 ,11 ,5] ⇒ 1
- [11, 45, 12, 10, 11] ⇒ 2

More recursion

- Insert * into between duplicate letters
- Input ⇒ String to insert
- Return ⇒ String with * between any duplicate letter

- “hello” ⇒ “hel*lo”
- “aabb” ⇒ “a*ab*b”
- “aaa” ⇒ “a*a*a”

More recursion

- Insert a value into a linked list
- Input ⇒ value and index
- Return ⇒ None, modify list

- We'll use this weeks LP code to start with

More recursion

- Find index of value
- Input ⇒ sorted array, target value
- Return ⇒ index of target value

- Special algorithm
- Binary search
- Very fast ⇒O(nlogn) time