LEC 08

CSE 123

Recursion

Questions during Class?
Raise hand or send here

sli.do #cse123



BEFORE WE START

Talk to your neighbors:

What is your favorite season/weather?

Respond on sli.do!

Instructor: Nathan Brunelle

TAs:

Cora	Chris	Crystal	Anya	Evan
Jonah	Nhan	Saachi	Ishita	Sahana
Chloe	Shiven	Nate	Renee	Yuntong
Amiya	Shashmit	Akshaya	Vihan	Shreedhaarini
Roshan	Adiyan	Nancy	Rachel	Ananya
Ryuta	Mimi	Isaac	Megan	

Lecture Outline: Announcements

- **Announcements**
- Method Calls Review
- Recursion Review
- Recursive Tracing Practice

Announcements

- Programming Project 1 released (Due Nov. 4)
- Computer Quiz 0 is happening!
  - Sign up on Canvas!
  - Your only days left are Monday 10/26 and Tuesday 10/27
- Paper Quiz 0 on Nov. 3 in section
  - Practice quiz coming next week
  - Covers Interfaces, Abstract Classes, Inheritance, polymorphism, Comparable, and ArrayLists
  - Does not cover ListNodes, LinkedLists

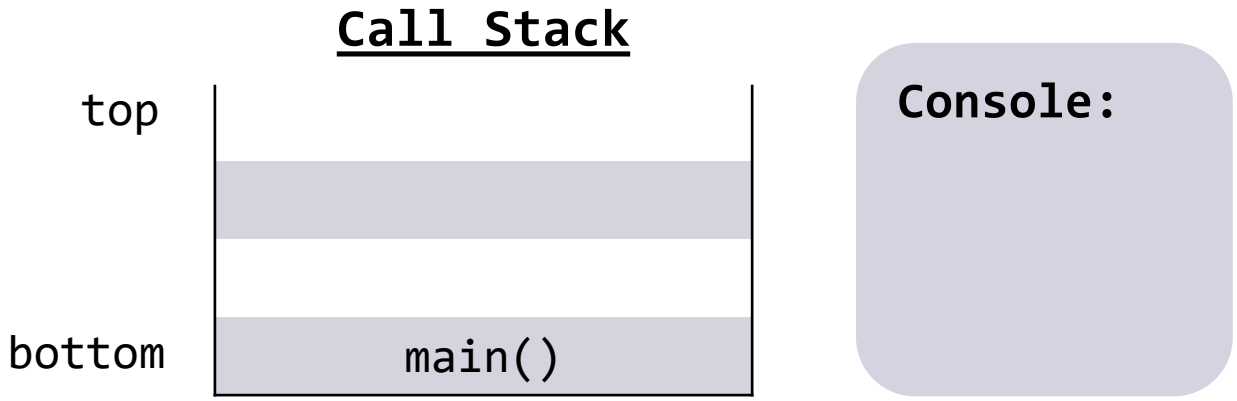
Lecture Outline: Method Calls Review

- Announcements
- **Method Calls Review**
- Recursion Review
- Recursive Tracing Practice

Method Calls

- Regardless how you use them, methods work the same way!
  - Pause execution, finish method, return where you left off
- How does Java keep track of the prior method?
  - Sort of like the "history" of the method calls it has paused… (LIFO)
  - Something called the **Call Stack**

Call Stack: Step 1



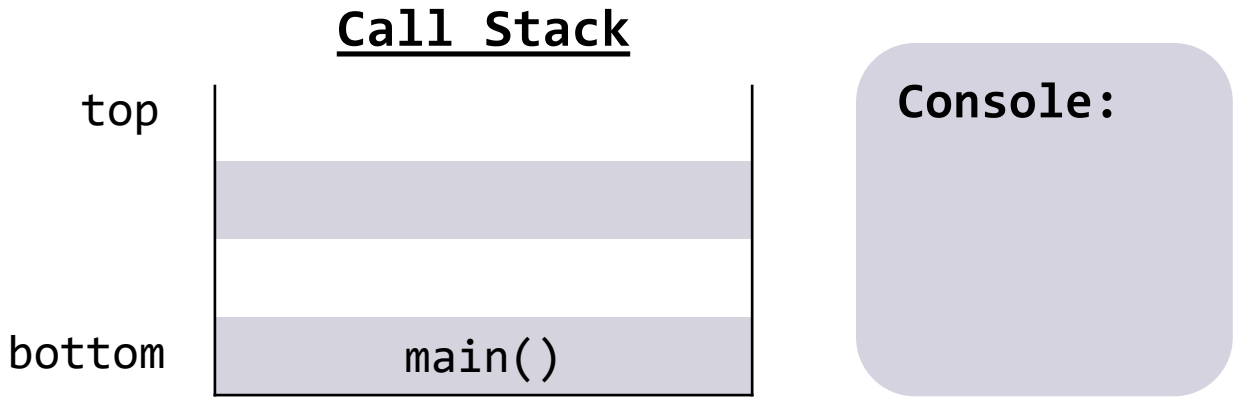

```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 2



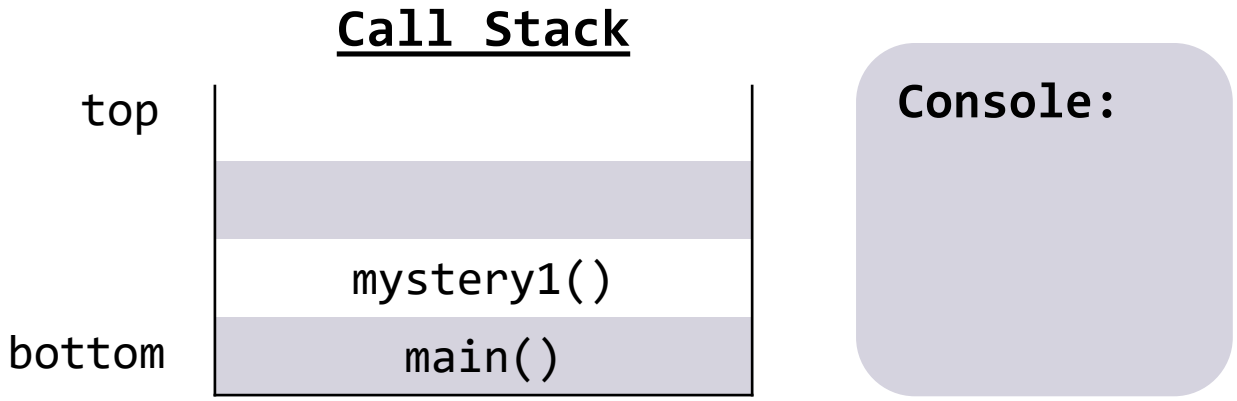

```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 3



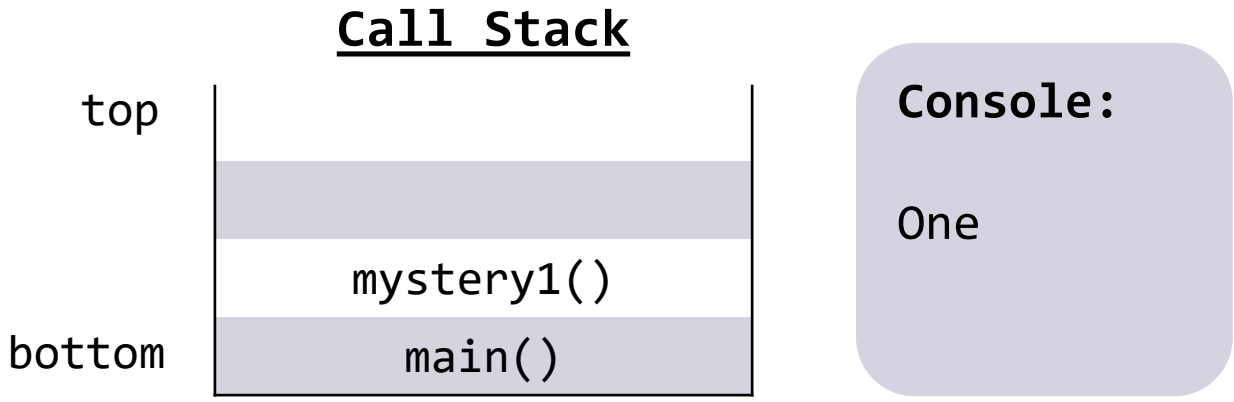

```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 4



Console:

One

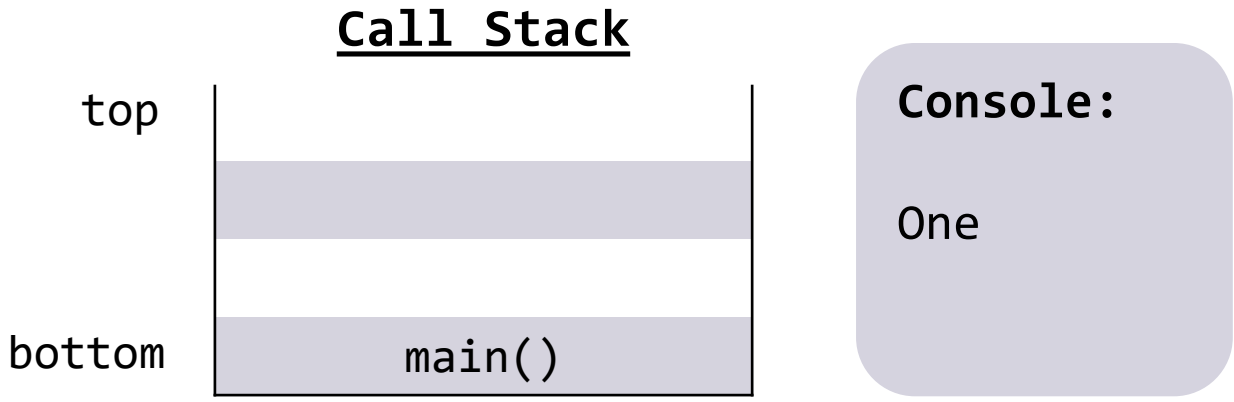
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 5



Console:

One

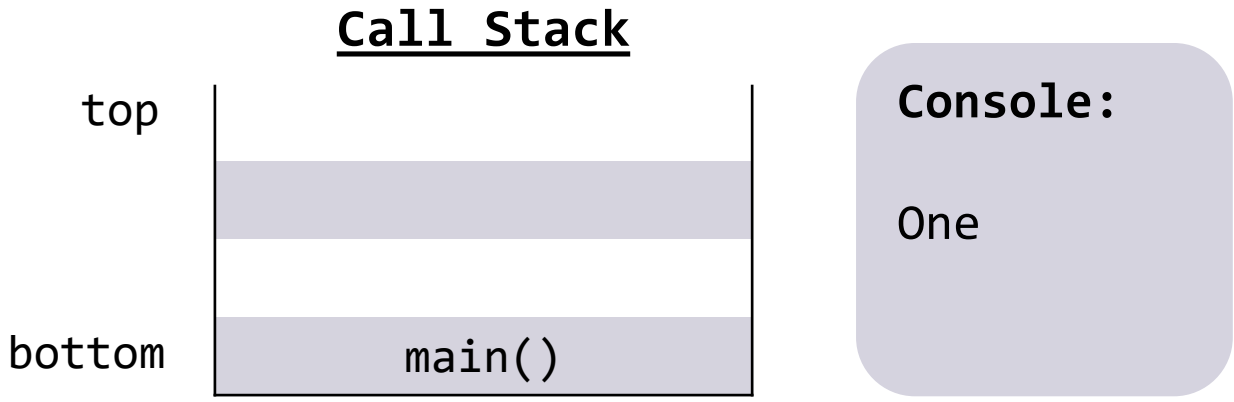
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 6



Console:

One

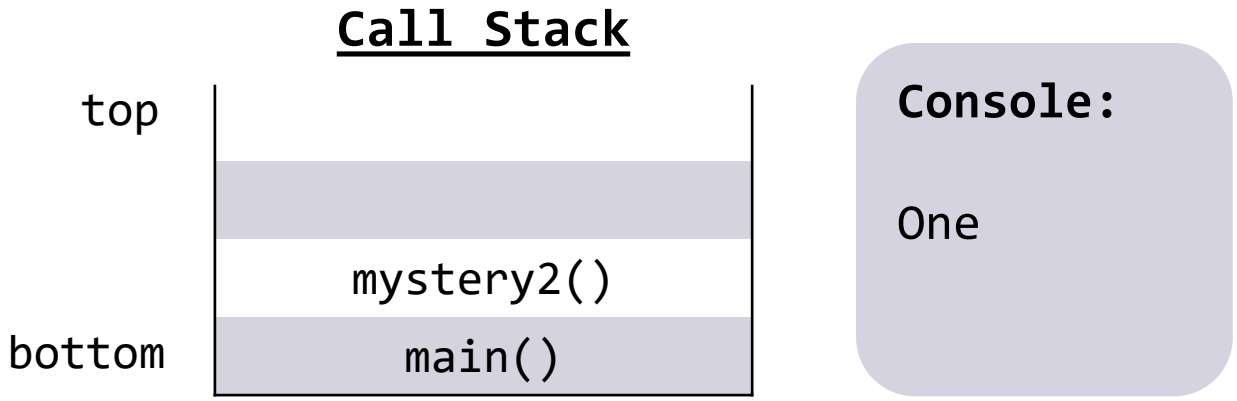
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 7



Console:

One

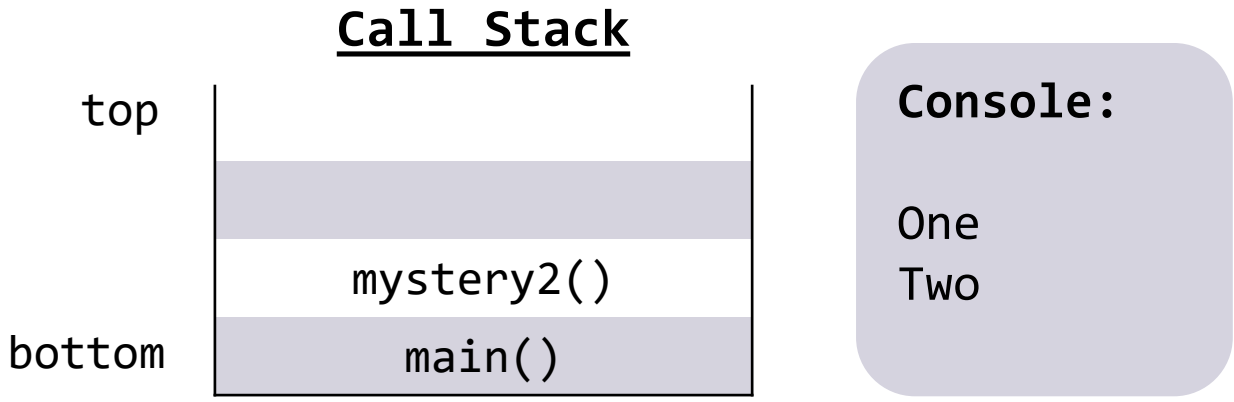
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 8



Console:

One
Two

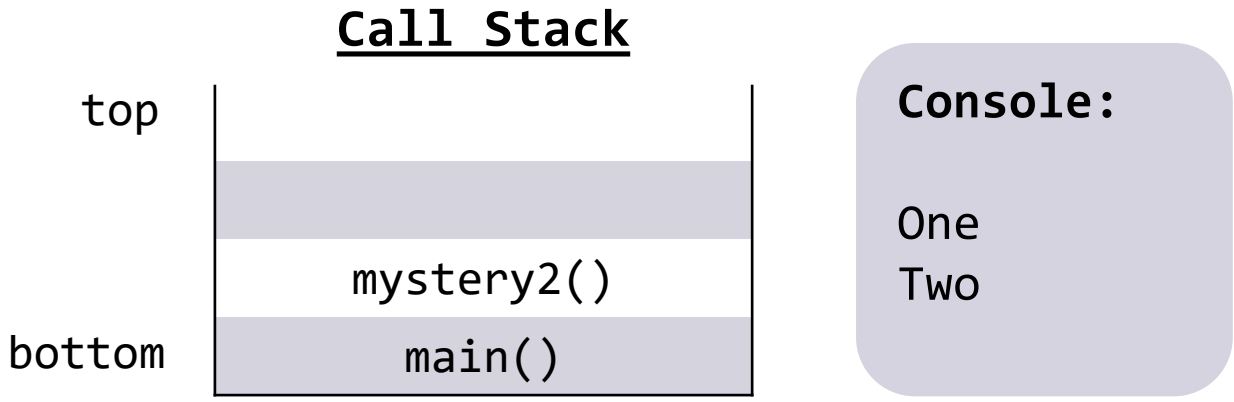
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 9



Console:

One
Two

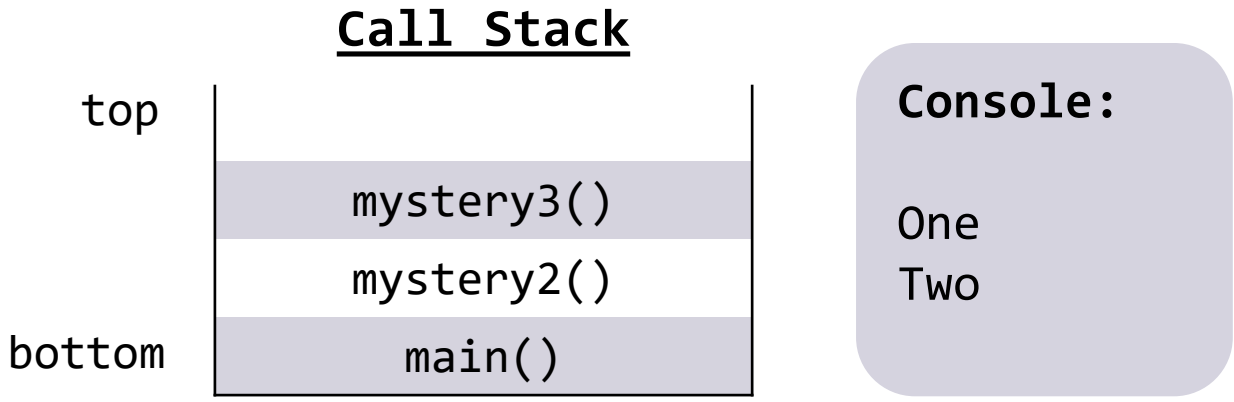
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 10



Console:

One
Two

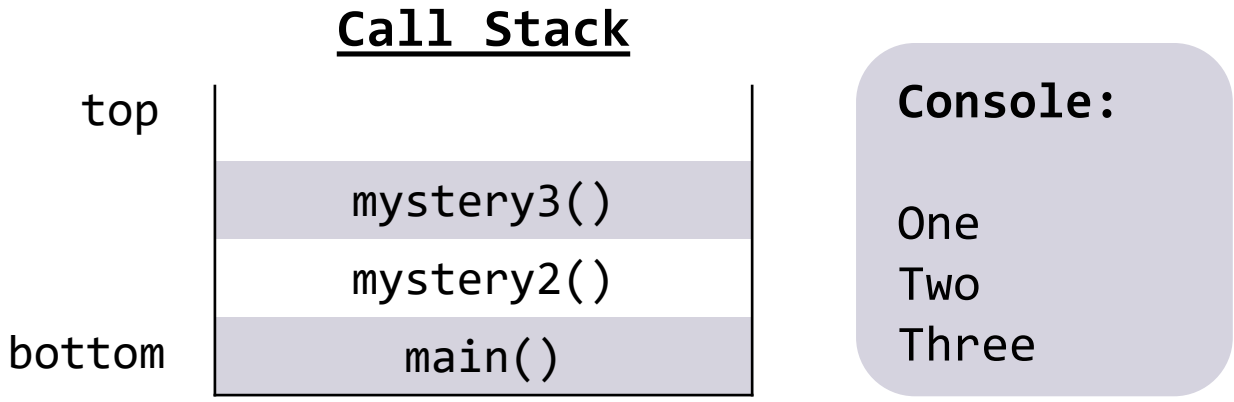
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 11



Console:

One
Two
Three

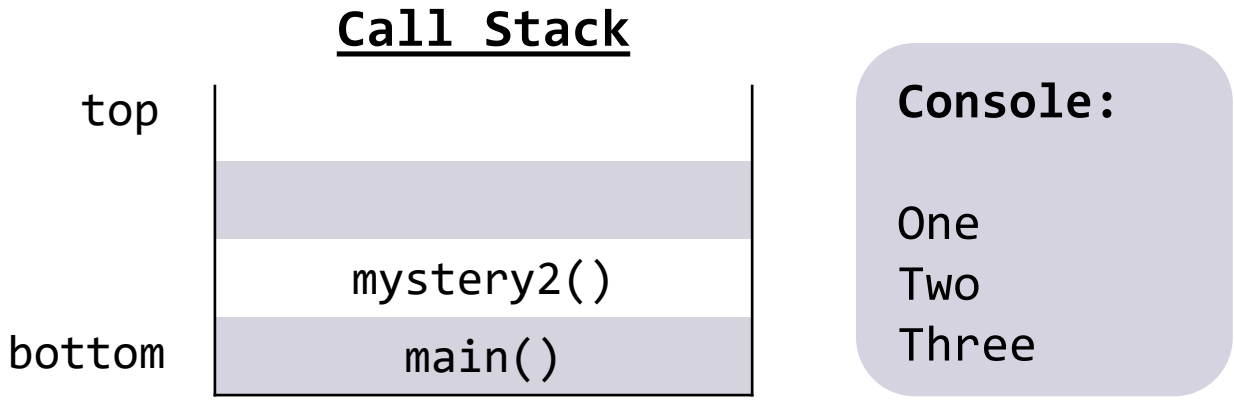
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 12



Console:

One
Two
Three

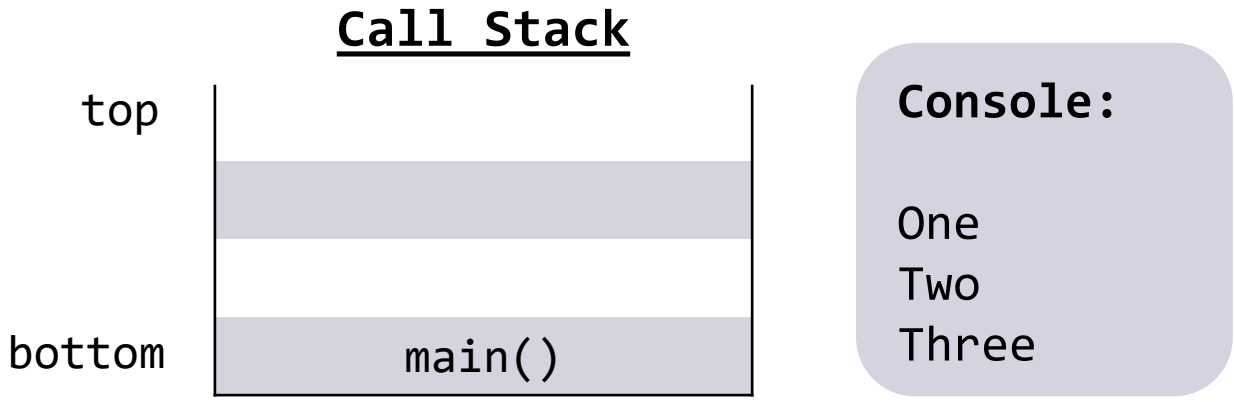
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 13



Console:

One
Two
Three

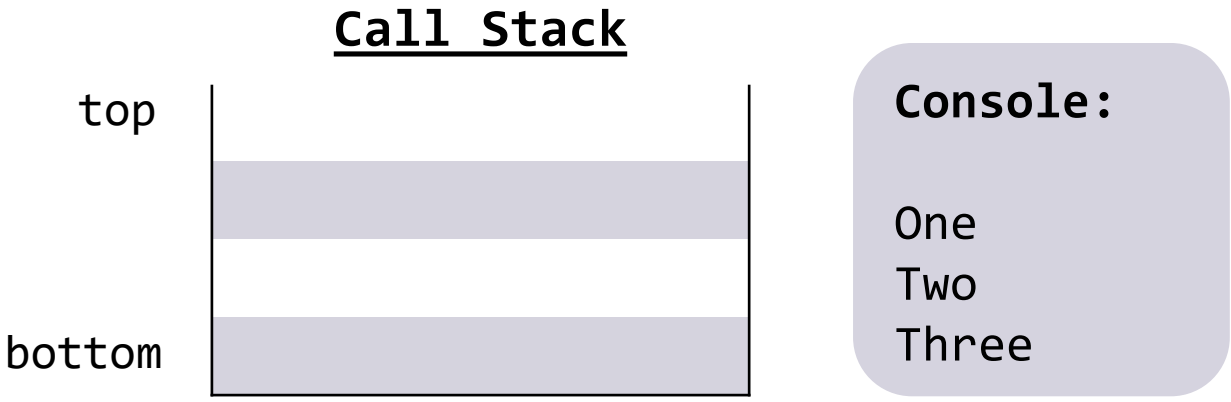
```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Call Stack: Step 14



Console:

One
Two
Three

```
public static void main(String[] args) {
    mystery1();
    mystery2();
}

public static void mystery1() {
    System.out.println("One");
}
```

```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

public static void mystery3() {
    System.out.println("Three");
}
```

Lecture Outline: Recursion Review

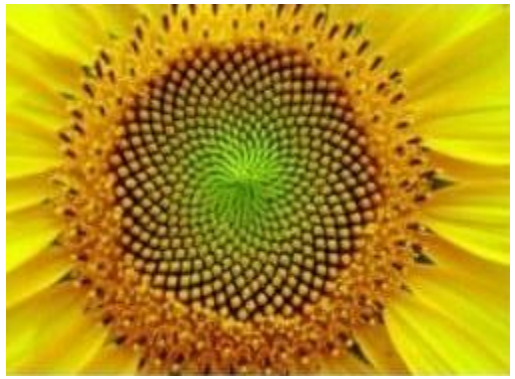
- Announcements
- Method Calls Review
- **Recursion Review**
- Recursive Tracing Practice

Recursion (1)

"The repeated application of a recursive procedure or definition"

- Oxford Languages

- Real-world definition: defining a problem in terms of itself
  - Case in point: above definition
  - Further natural examples:






Recursion (2)

"The repeated application of a recursive procedure or definition"

- Oxford Languages

- Real-world definition: defining a problem in terms of itself
  - Case in point: above definition
  - Further natural examples:



Recursion (3)

"The repeated application of a recursive procedure or definition"

- Oxford Languages

- Real-world definition: defining a problem in terms of itself
  - Case in point: above definition

- Computer science definition: when a method calls itself
  - "Alternative" to iteration (can combine for powerful results)

Recursion (4)

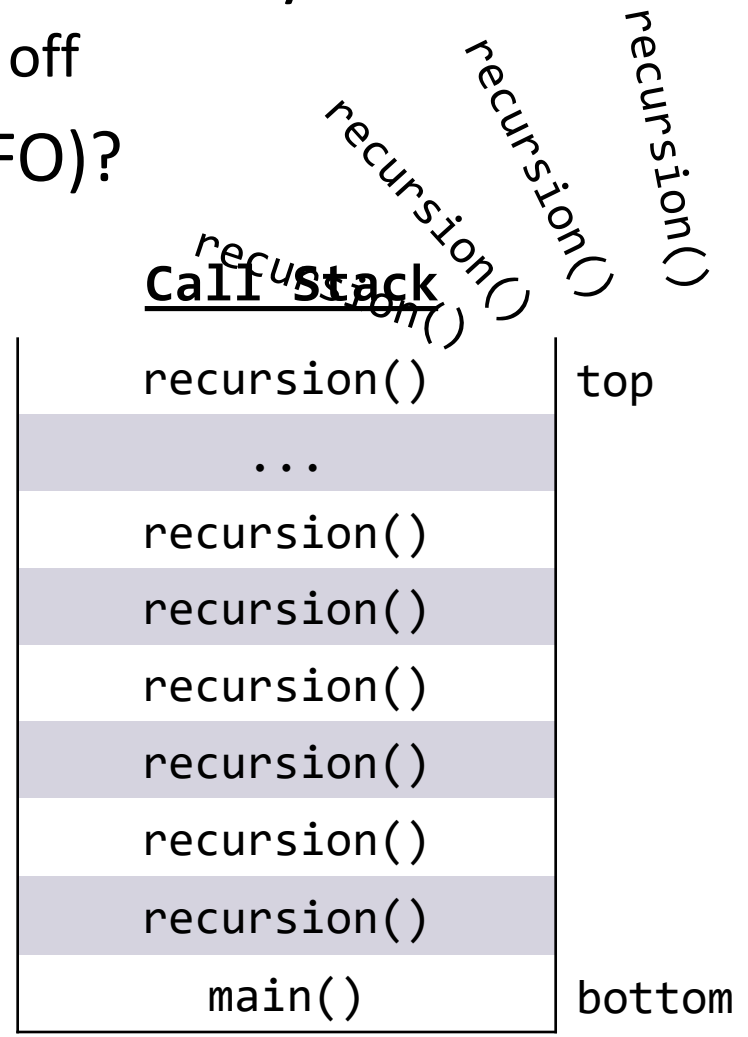
"The repeated application of a recursive procedure or definition"

- Oxford Languages

- Why do we care?
  - It's a fundamental Computer Science topic
  - Some things are easier to describe with recursion than with loops
  - There are some problems that lend themselves more easily to a recursive solution than an iterative (loop-y) solution
    - Though all problems that can be solved with recursion can be solved with loops, and vice versa

Method Calls

- Regardless how you use them, methods work the same way!
  - Pause execution, finish method, return where you left off
- How does Java keep track of the prior method (LIFO)?
  - Something called the **Call Stack**



🤔 Wouldn't that just lead to an infinite loop?

```
public static void recursion() {
    System.out.println("Woah");
    recursion();
}
```

StackOverflowException!

Method Calls: Avoiding Infinite Recursion

- Regardless how you use them, methods work the same way!
  - Pause execution, finish method, return where you left off
- How does Java keep track of the prior method (LIFO)?
  - Something called the **Call Stack**

🤔 Wouldn't that just lead to an infinite loop?
  - Yes! We get something called a **StackOverflowException**

- How do we avoid infinite recursion?

Recursive Methods

- 2 components of every recursive method:

- Recursive case
  - Decompose problem into subproblem
  - Make the actual recursive call
  - Combine results meaningfully
- Base case
  - Simplest version of the problem
  - No subproblems to break into
  - Return known answer



Recursive Methods: Decomposition

- 2 components of every recursive method:

- Recursive case
  - Decompose problem into subproblem
  - Make the actual recursive call
  - Combine results meaningfully
- Base case
  - Simplest version of the problem
  - No subproblems to break into
  - Return known answer



If decomposing moves you closer to the base, no infinite recursion!

Math Examples

n! or factorial(n) = product of all positive integers <= n

- Two ways of viewing this idea:

- $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$
  - Iterative approach - loop through all values and multiply together

- $n! = n * (n-1)!$
  - Recursive approach – decompose into subproblem and combine
  - What would our base case / simplest input (n) be?

Math Examples: Hitting the Base Case

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n-1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2!$



$2! = 2 * 1!$



$1! = 1 * 0!$

$0! = 1$

Math Examples: Popping off callstack 1

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n-1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2!$



$2! = 2 * 1!$



$1! = 1 * 0!$



$0! = 1$

Math Examples: Popping off callstack 2

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2!$



$2! = 2 * 1!$



$1! = 1 * 1$

Math Examples: Popping off callstack 3

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2!$



$2! = 2 * 1!$



$1! = 1$

Math Examples: Popping off callstack 4

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2!$



$2! = 2 * 1$

Math Examples: Popping off callstack 5

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2!$



$2! = 2$

Math Examples: Popping off callstack 6

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n-1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 3 * 2$

Math Examples: Popping off callstack 7

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 4 * 3!$



$3! = 6$

Math Examples: Popping off callstack 8

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 4 * 6$

Math Examples: Popping off callstack 9

n! / factorial(n) = product of all positive integers <= n

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example



$4! = 24$

Lecture Outline: Recursive Tracing Practice

- Announcements
- Method Calls Review
- Recursion Review
- **Recursive Tracing Practice**