LEC 08

CSE 123

Recursion

BEFORE WE START

Talk to your neighbors:

What's your favorite icebreaker question? 🤔

Instructor: James Wilcox

Questions during Class?

Raise hand or send here

sli.do #cse123

Announcements

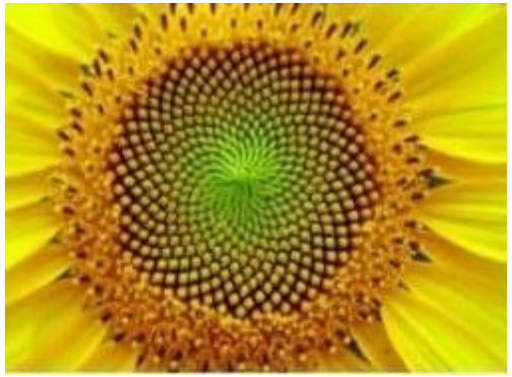
- Quiz 1 Tuesday!
  - Topics: Abstract classes, Implementing data structures (ArrayLists / LinkedLists), Runtime
  - Topics not on Quiz: Recursion
  - Practice quiz is available!
- Quiz 0 feedback will be released Friday afternoon
- Programming Assignment 1 due in one week (10/30) @ 11:59pm
- Resubmission Period 3 closes this Friday (10/25) @ 11:59pm
  - Available assignments: **C0**, P0, C1
  - Last opportunity to resubmit C0
- Reminder: Grade guarantee calculator

Recursion

"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

"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

"The repeated application of a recursive procedure or definition"

- Oxford Languages

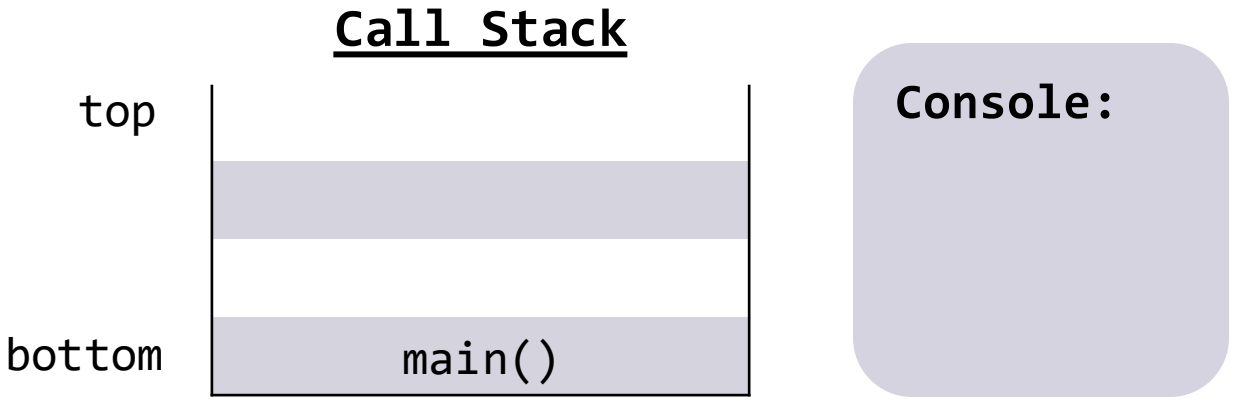
- Real-world definition: defining a problem in terms of itself
  - Case in point: above definition

- Computer science definition: when a method calls itself
  - "Alternative" to iteration (can combine for powerful results)
- 🤔 Wouldn't that just lead to an infinite loop?
  - Yes! If you do things wrong…
  - Let's review *how* method calls work

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**

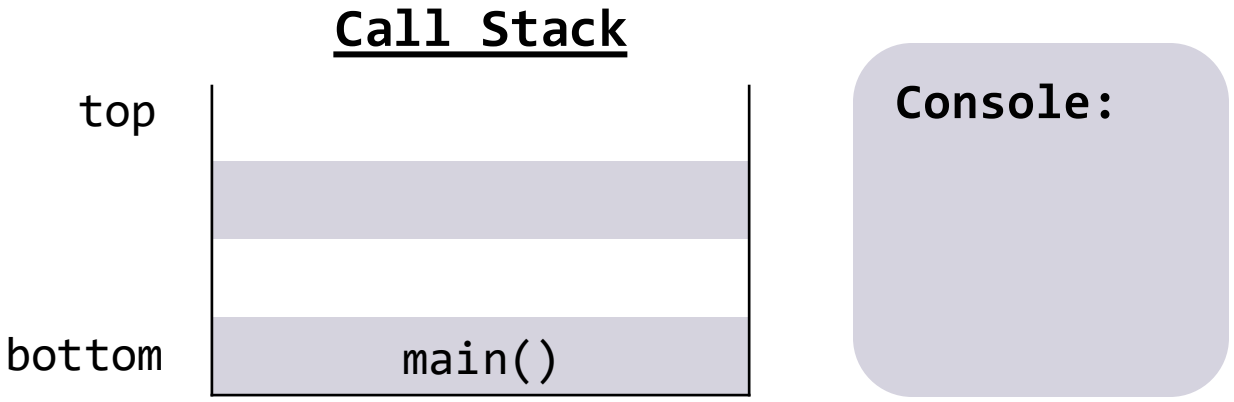
Call Stack




```
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");
}
```

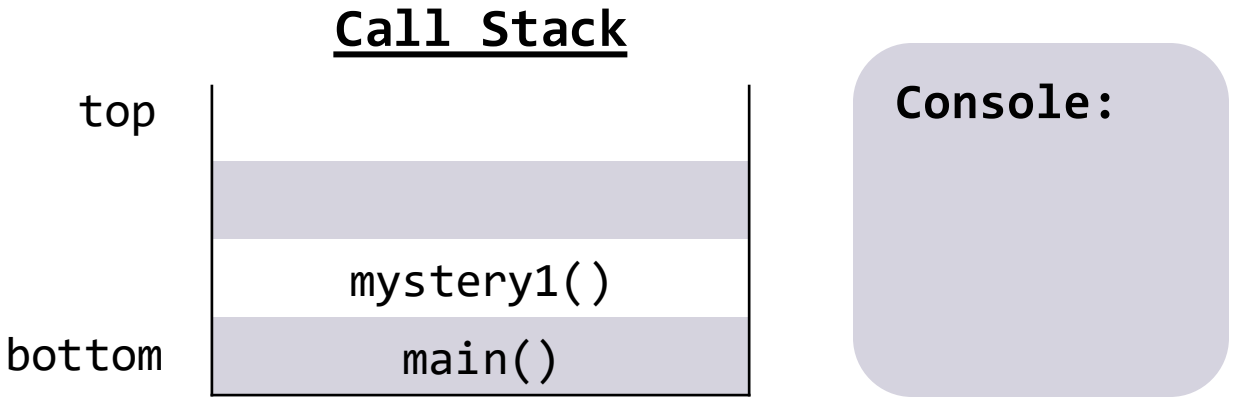
Call Stack




```
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");
}
```

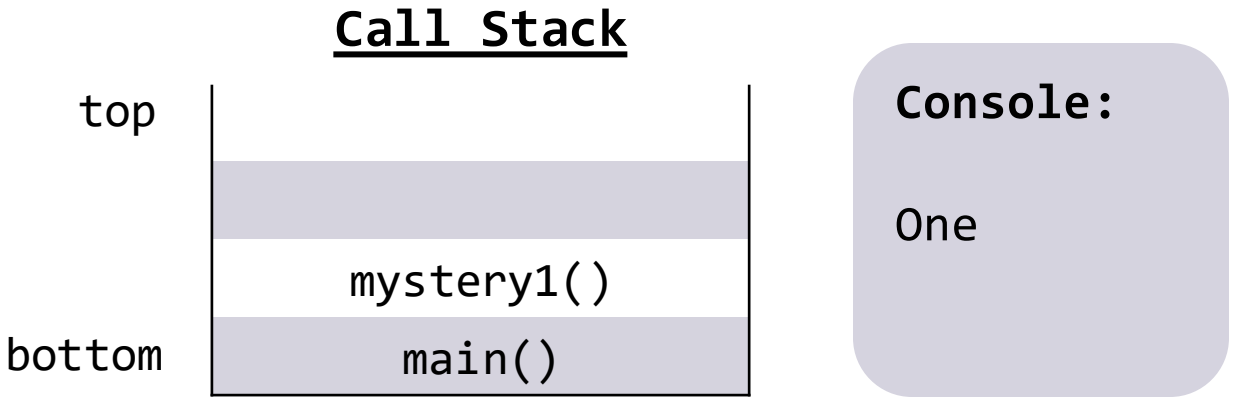
Call Stack




```
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



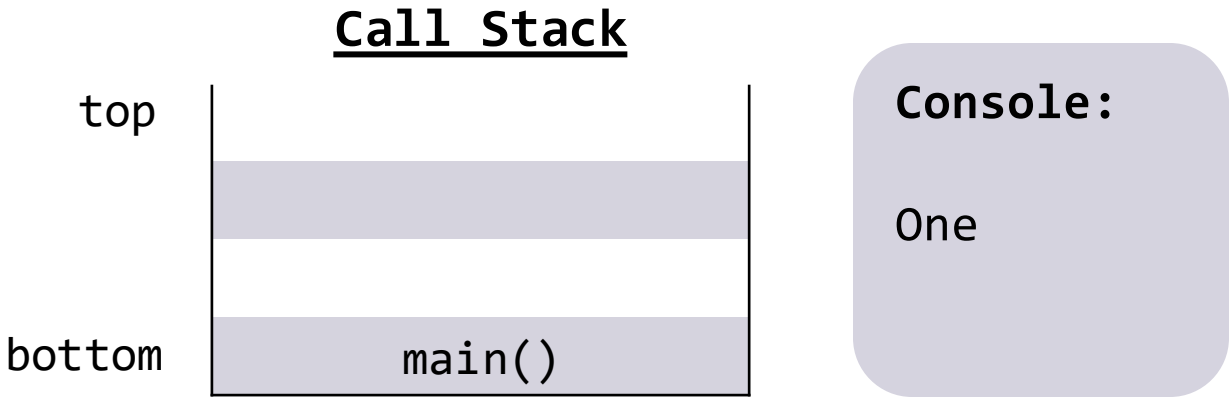
Console:

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



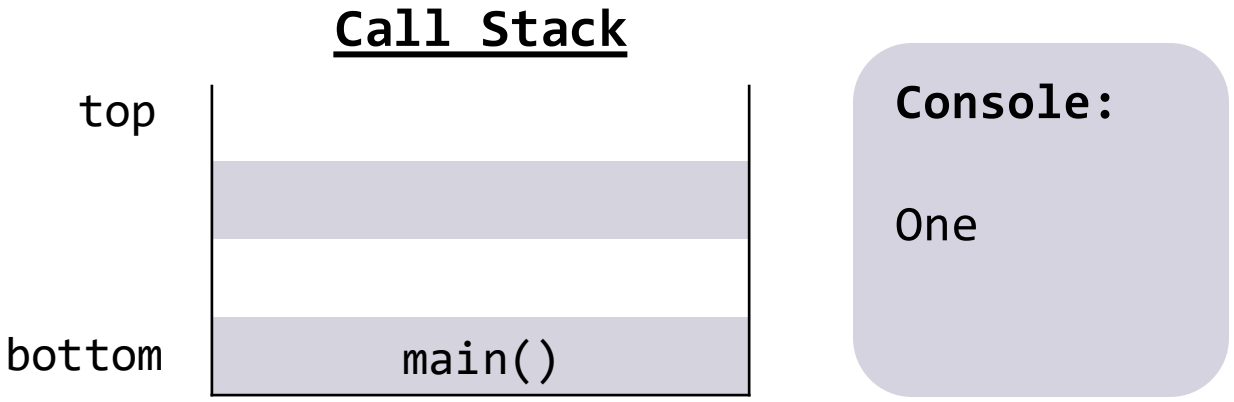
Console:

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



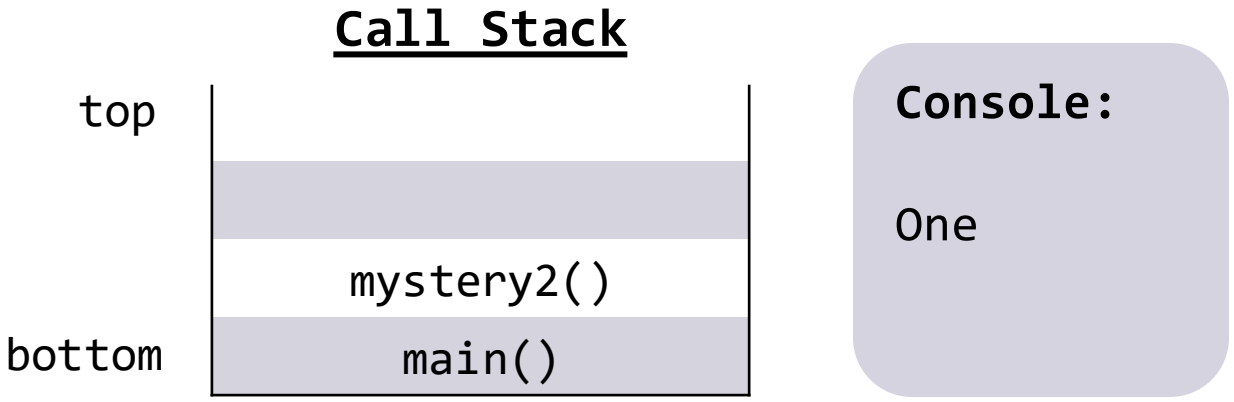
Console:

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");
}
```

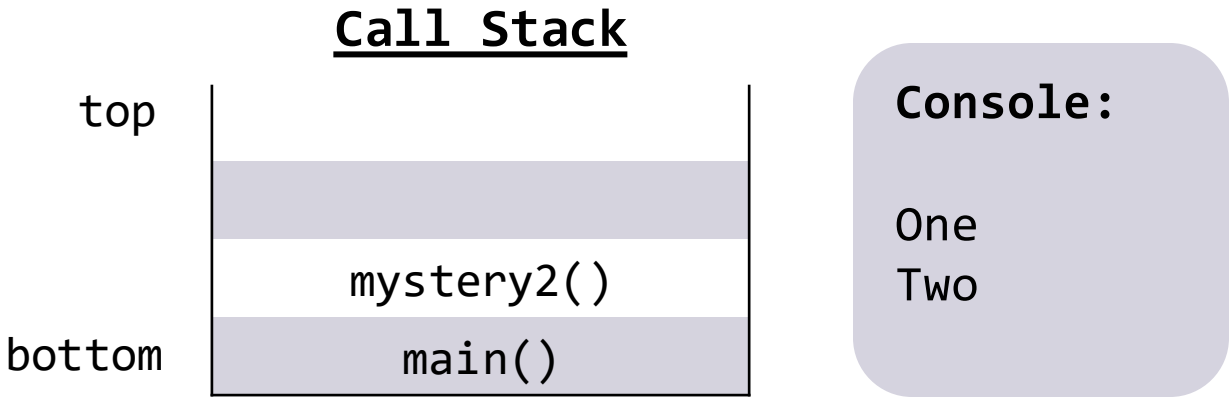
Call Stack




```
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



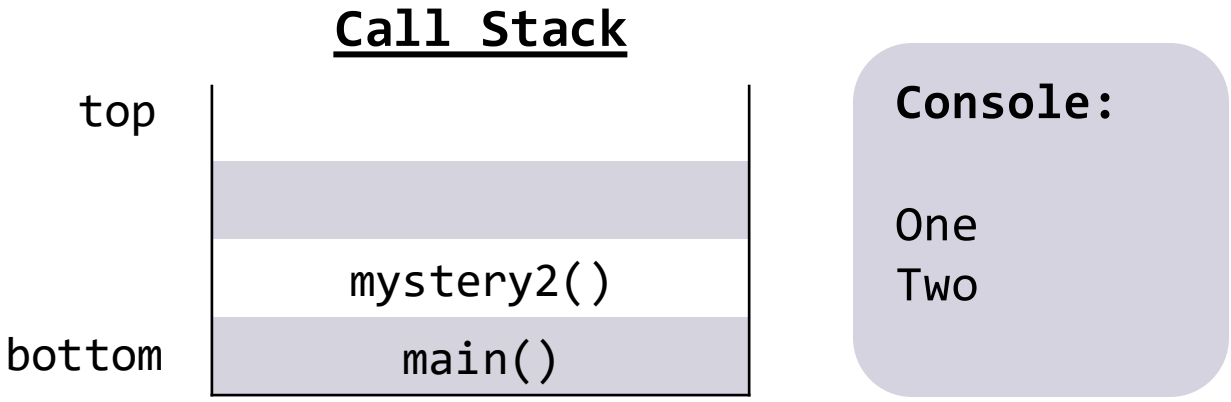
Console:

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



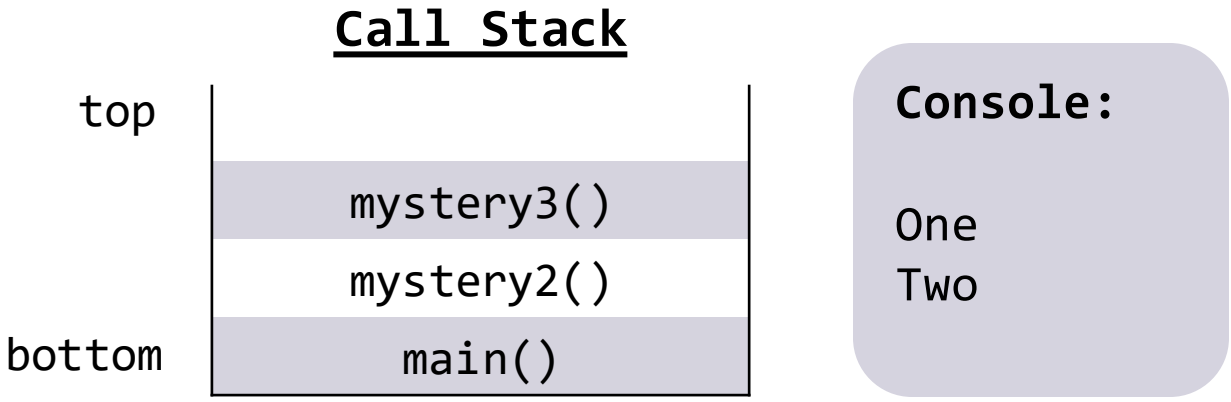
Console:

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



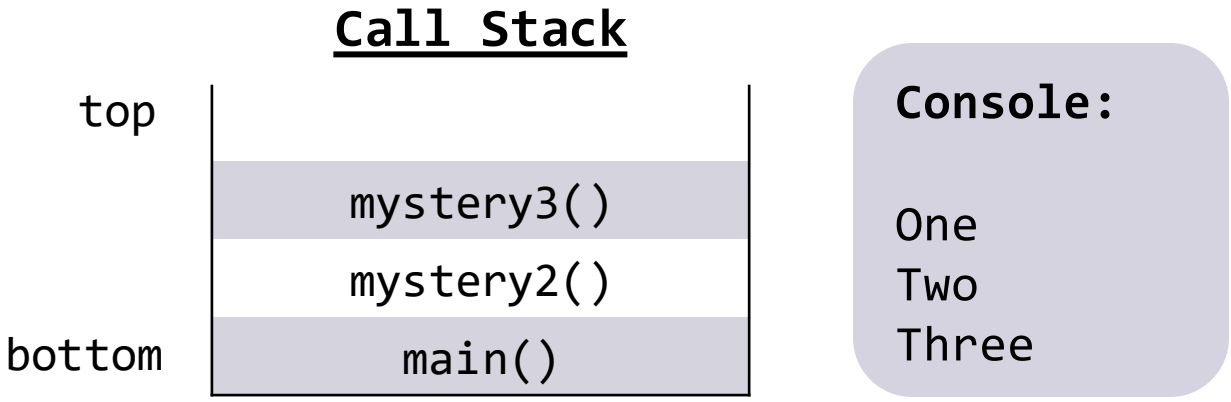
Console:

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



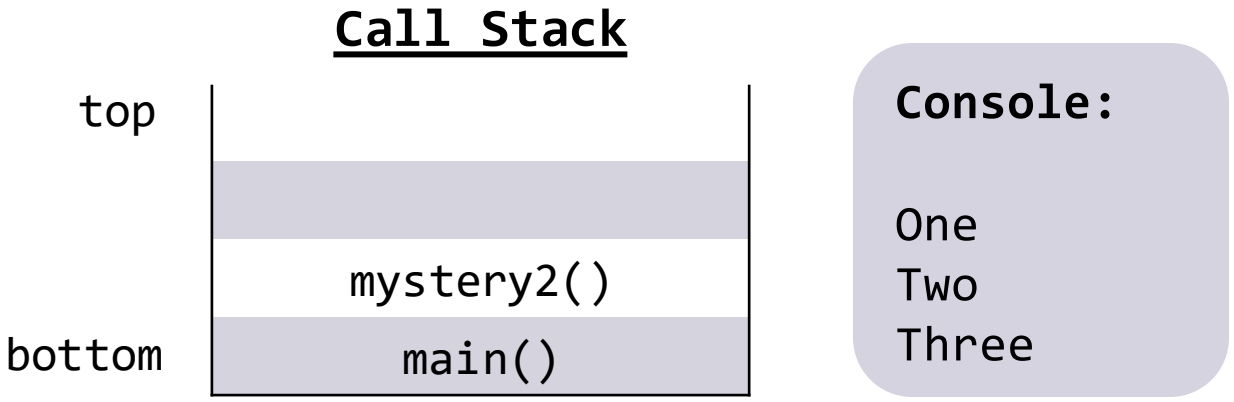
Console:

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



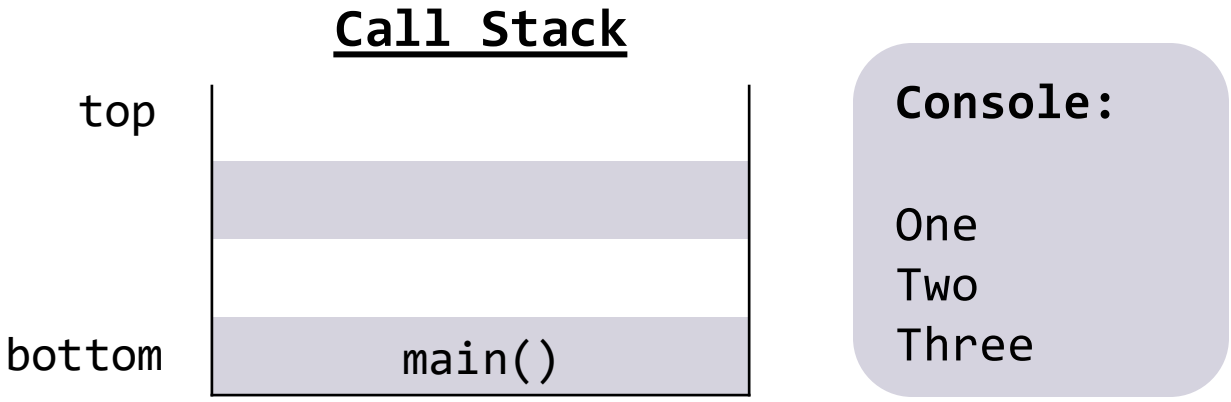
Console:

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



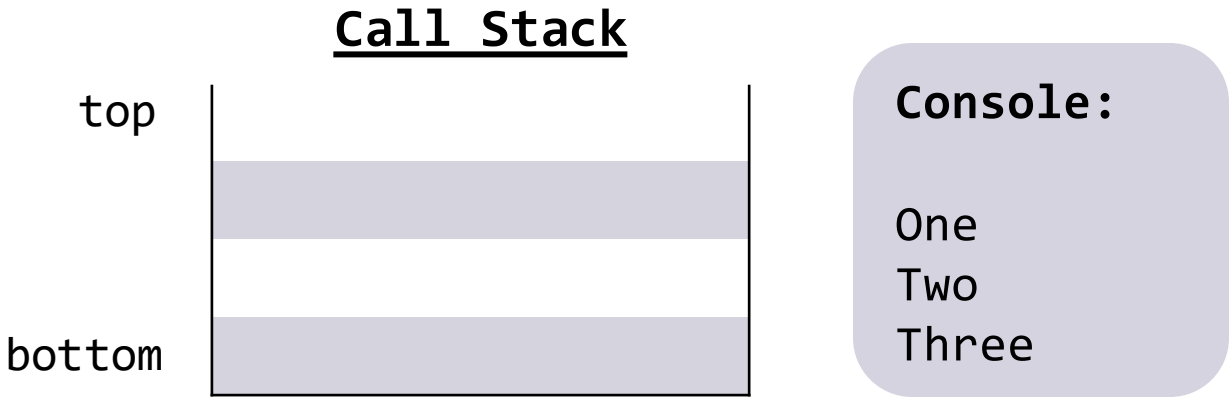
Console:

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



Console:

One
Two
Three

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

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

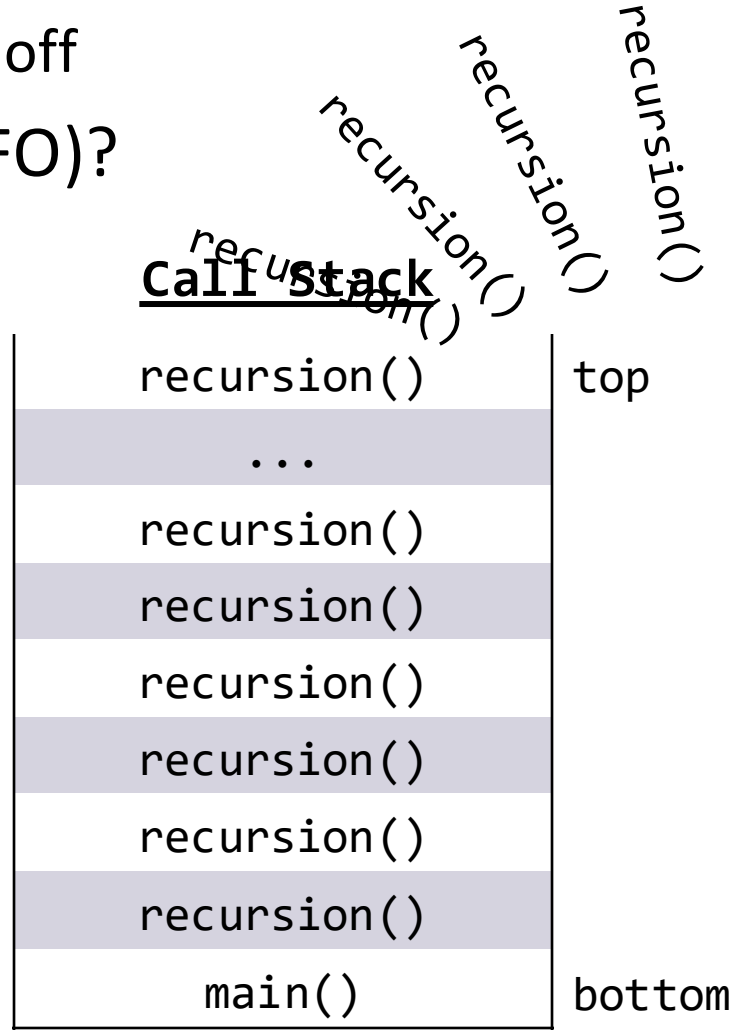
```
public static void mystery2() {
    System.out.println("Two");
    mystery3();
}

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

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();
}
```



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?

- 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

- 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! / 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

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

$0! = 1$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

$0! = 1$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 1$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$ $3! = 3 * 2$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 3!$

$3! = 6$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 4 * 6$

Math Examples

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

- Reminder, recursive approach: $n! = n * (n - 1)!$
  - Let's trace through with a simple example 3!

$4! = 24$