

Calls All the Way Down

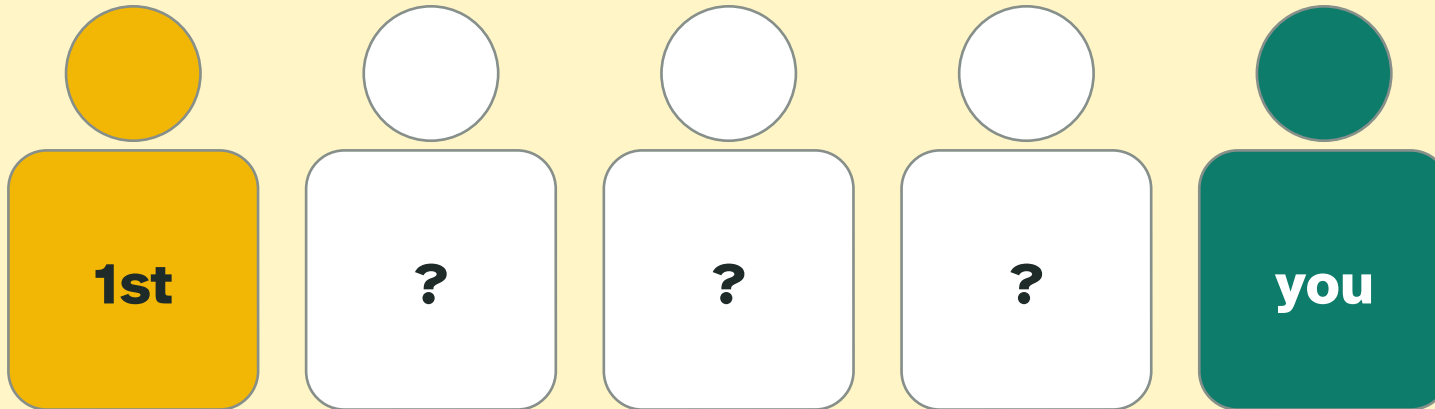
Recursion



What place are you in line?

PAUSE AND TRY

You can only talk to the person right in front of you. How can you find out your place in line?



Talk with a partner. What do you ask? Who can answer without asking anyone?

front of the line

By the end of class you can...

- 1 Find the base case and the recursive call in a recursive method.
- 2 Explain what makes a recursion stop, and spot one that never does.
- 3 Trace recursive calls on a call stack, where each call keeps its own variables.
- 4 Find what a recursive method returns by working back up the calls.
- 5 Compare recursion with loops, and explain why each can replace the other.

A large white square with rounded corners, tilted slightly to the right, containing a large, bold, dark teal number '1' in the center.

1

PART 1

A method that calls itself

The two parts of every recursive method

COUNTDOWN, CALLED WITH COUNTDOWN(3)

```
public static void countdown(int n) {  
    if (n == 0) {  
        System.out.println("Liftoff!");  
    } else {  
        System.out.println(n);  
        countdown(n - 1);  
    }  
}
```

OUTPUT

```
3  
2  
1  
Liftoff!
```

Base case

A path that finishes without calling again. Here: `n == 0` prints Liftoff! and stops.

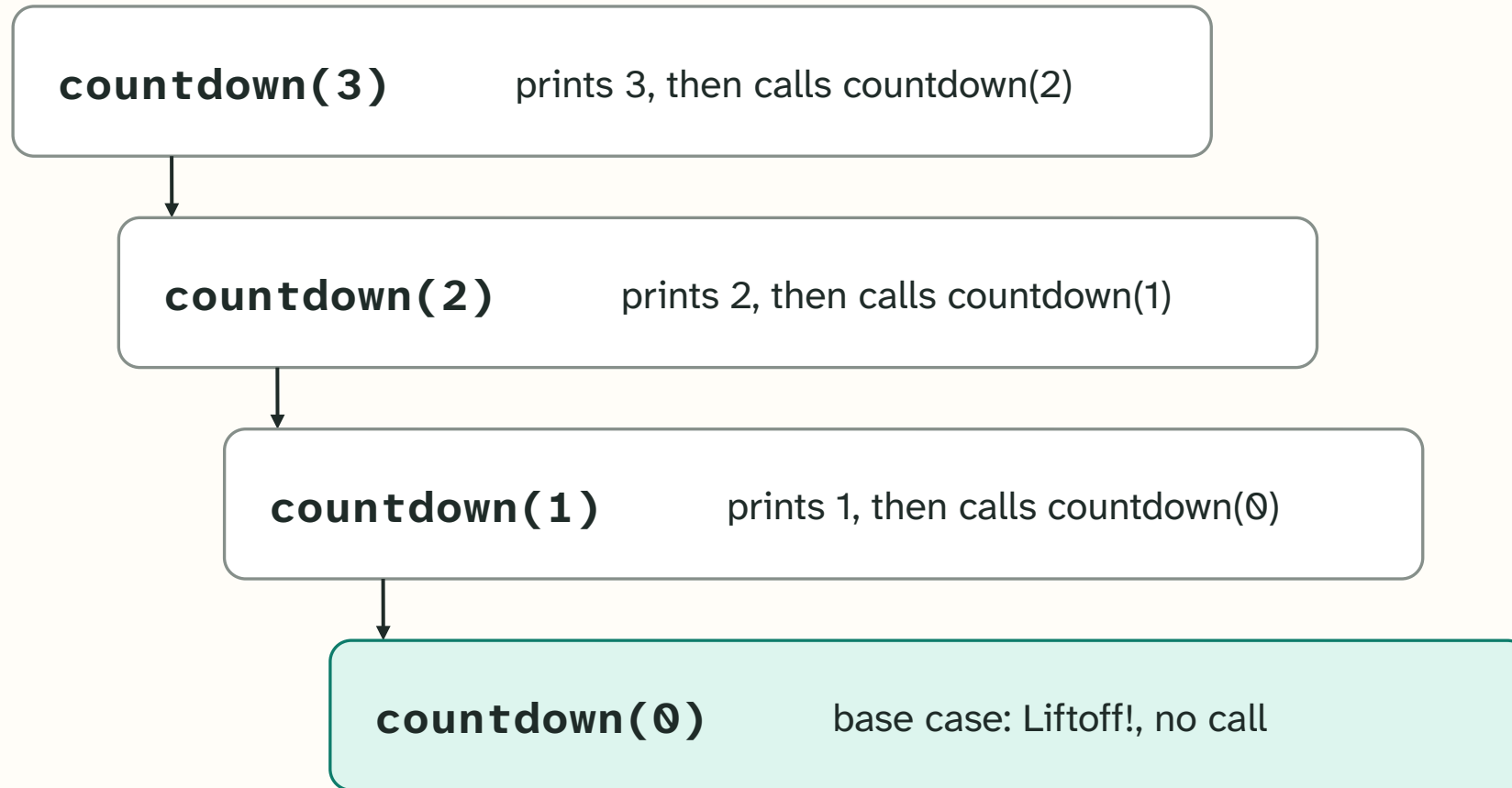
Recursive call

The method calls itself on a smaller job. Here: `countdown(n - 1)`.

Another kind of repetition

Every recursive method needs at least one of each. Together they repeat work, like a loop.

Tracing countdown(3), call by call



4 calls in all: one from main, then 3 the method makes itself.

The parameter shrinks by 1 each call, so it must reach **0**, the base case.

A base case you can jump over



It **compiles**. Java checks types, not whether the calls ever reach a base case. If they never do, it crashes with a **StackOverflowError**.

COUNTDOWN(3), NOW WITH N - 2

```
public static void countdown(int n) {  
    if (n == 0) {  
        System.out.println("Liftoff!");  
    } else {  
        System.out.println(n);  
        countdown(n - 2);  
    }  
}
```

OUTPUT

```
3  
1  
-1  
-3  
-5  
...
```

Exception in thread "main"
java.lang.StackOverflowError

Which starts reach the base case?

PAUSE AND TRY

For each start, does step(start) print done, or crash?

```
public static void step(int n) {  
    if (n == 12) {  
        System.out.println("done");  
    } else {  
        step(n + 3);  
    }  
}
```

Start	done, or crash?
step(0)	
step(3)	
step(5)	
step(12)	
step(15)	

A large, bold, teal-colored number '2' is centered within a white rounded square that has a slight 3D effect and a shadow.

2

PART 2

Each call has its own variables

KEY IDEA

The call stack: newest call on top

CALLED WITH STAIRS(3)

```
public static void stairs(int n) {  
    if (n > 0) {  
        stairs(n - 1);  
        System.out.println("step " + n);  
    }  
}
```

OUTPUT

```
step 1  
step 2  
step 3
```

Each call keeps its own n. Calls finish newest first.

CALL STACK AT ITS TALLEST

stairs(0) $n = 0$ base case: returns at once

stairs(1) $n = 1$ waiting, then prints step 1

stairs(2) $n = 2$ waiting, then prints step 2

stairs(3) $n = 3$ waiting, then prints step 3

In, middle, out

PAUSE AND TRY

Write the exact output of visit(2).

```
public static void visit(int n) {  
    if (n == 0) {  
        System.out.println("middle");  
    } else {  
        System.out.println("in " + n);  
        visit(n - 1);  
        System.out.println("out " + n);  
    }  
}
```

YOUR OUTPUT

Hint: which line runs before the call,
and which runs after it returns?

A large, bold, teal number '3' is centered within a white rounded square that has a slight 3D effect and a shadow.

PART 3

Returning values on the way back

WORKED EXAMPLE

Going down, coming back up

CALLED WITH SUMTO(4)

```
public static int sumTo(int n) {  
    if (n == 1) {  
        return 1;  
    }  
    return n + sumTo(n - 1);  
}
```

OUTPUT

10

Call	Going down	Back up: returns
sumTo(4)	4 + sumTo(3)	4 + 6 = 10
sumTo(3)	3 + sumTo(2)	3 + 3 = 6
sumTo(2)	2 + sumTo(1)	2 + 1 = 3
sumTo(1)	base case	1

Each call combines **its own work** with the value its recursive call **returns**.

Trace it: going down, then back up

PAUSE AND TRY

What does `mystery(14)` return? How many calls are made in all?

```
public static int mystery(int n) {  
    if (n < 3) {  
        return n;  
    }  
    return mystery(n / 3) + n % 3;  
}
```

Call	Waits on	Returns
mystery(14)		

A large, white, rounded square containing the number '4' in a bold, teal, sans-serif font.

PART 4

Recursion and loops

The parameter does the job of the loop variable

LOOP

```
for (int n = 3; n > 0; n--) {  
    System.out.println(n);  
}  
System.out.println("Liftoff!");
```

Both programs print the same thing:

OUTPUT

```
3  
2  
1  
Liftoff!
```

RECURSION, CALLED WITH COUNTDOWN(3)

```
public static void countdown(int n) {  
    if (n == 0) {  
        System.out.println("Liftoff!");  
    } else {  
        System.out.println(n);  
        countdown(n - 1);  
    }  
}
```

Job	Loop	Recursion
Starts at	int n = 3	countdown(3)
Stops when	n > 0 is false	base case n == 0
Moves on by	n - -	countdown(n - 1)

Either one can replace the other

ITERATIVE, CALLED WITH SUMLOOP(5)

```
public static int sumLoop(int n) {  
    int total = 0;  
    for (int i = n; i > 0; i--) {  
        total += i;  
    }  
    return total;  
}
```

OUTPUT

15

RECURSIVE, CALLED WITH SUMREC(5)

```
public static int sumRec(int n) {  
    if (n == 0) {  
        return 0;  
    }  
    return n + sumRec(n - 1);  
}
```

OUTPUT

15

Any recursive solution can be written **iteratively**, and any iterative one **recursively**.

Words to know

Recursive method

A method that calls itself.

Base case

A path that finishes without a recursive call, halting the recursion.

Recursive call

A call to a method from inside that same method.

Call stack

The calls that have started but not finished, newest on top.

StackOverflowError

The crash when waiting calls use up the stack, usually from an unreachable base case.

Iterative

Repeating with a loop instead of recursion.

How does it ever stop?



Explain to your partner how a method that calls itself ever stops. Use countdown as your example.

SENTENCE STARTER

The recursion stops when _____, and every recursive call has to _____.

Next: the Calls All the Way Down puzzle worksheet.