Recursive Methods

Sometimes when solving a problem, we can compute the solution of a simpler version of the *same problem*. Eventually we reach the most basic version, for which the answer is trivial.

Manager:		Recorder:	
Presenter:		Reflector:	

Content Learning Objectives

After completing this activity, students should be able to:

- Identify the base case and recursive step of the factorial method.
- Trace a recursive method by hand and predict its final output.
- Explain what happens in memory when a method calls itself.

Process Skill Goals

During the activity, students should make progress toward:

- Evaluating mathematical sequences to gain insight on recursion. (Information Processing)

Facilitation Notes

This activity is a first introduction to recursion using two mathematical examples: factorial and summation. Students learn how to read and trace recursive methods, not how to formulate recursive solutions to problems. Let students wrestle with these difficult concepts, and don't give too much help on individual questions.

Keep an eye on questions 1–3, and if a team is getting off track, have them compare answers with a neighboring team. You may need to point out that ! in mathematics means ***factorial***, but ! in Java means ***not***. It's unfortunately common that operators have slightly different meanings in different languages.

Report out on #5 and #8 by having teams write their answers on the board. Ideally there will be slightly different (incorrect) answers, which will lead to a discussion. Paste *Recursion.java* into Java Visualizer and step through the code as a live demo. After #9, introduce the term ***stack overflow*** and make the connection to the ***call stack*** visualization in Java Visualizer.

Depending on how long you report out, Model 1 could take an additional 5–10 minutes. Model 2 should move a bit faster, since summation is almost identical to factorial. Invite presenters to write their team's solution to #16 on the board. Address any misconceptions about variables, parameter passing, and return values.

Key questions: #6, #9, #16

Source files: *Recursion.java*

Model 1 Factorial

"In mathematics, the ***factorial*** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$."

Source: https://en.wikipedia.org/wiki/Factorial

n	$n!$
0	1
1	1
2	2
3	6
4	24
5	120

Questions (25 min) **Start time:**

1. Consider how to calculate $4! = 24$.

a) Write out all the numbers that need to be multiplied:

$4! =$ 4 * 3 * 2 * 1

b) Rewrite the expression using 3! instead of $3 \times 2 \times 1$:

$4! =$ 4 * 3!

2. Write an expression similar to #1b showing how each factorial can be calculated in terms of a simpler factorial.

a) $3! =$ 3 * 2!

b) $2! =$ 2 * 1!

c) $100! =$ 100 * 99!

d) $n! =$ $n * (n-1)!$

3. What is the value of 0! based on Model 1? Does it make sense to define 0! in terms of a simpler factorial? Why or why not?

0! is 1 by convention for an empty product. We can't say $0 \times -1!$, because factorial is only defined for non-negative integers. At some point we need to define the solution in concrete terms, without referencing itself.

If we repeatedly break down a problem into smaller versions of itself, we eventually reach a basic problem that can't be broken down any further. Such a problem, like 0!, is referred to as the ***base case****.*

4. Assume you already have a working method named `factorial(int n)` that returns $n!$ for any positive integer.

a) Review your answer to #2c that shows how to compute 100! using a simpler factorial. Convert this expression to Java by using the `factorial` method instead of the `!` operator.

```
100 * factorial(99)
```

b) Now rewrite your answer to #2d in Java using the variable `n`.

```
n * factorial(n - 1)
```

5. Here is a `factorial` method that includes output for debugging:

```
1  public static int factorial(int n) {
2      System.out.println("n is " + n);
3      if (n == 0) {
4          return 1;  // base case
5      } else {
6          System.out.printf("need factorial of %d\n", n - 1);
7          int answer = factorial(n - 1);
8          System.out.printf("factorial of %d is %d\n", n - 1, answer);
9          return n * answer;
10     }
11 }
12
13 public static void main(String[] args) {
14     System.out.println(factorial(3));
15 }
```

a) What specific method is invoked on line 7?

The `factorial` method invokes itself (with a smaller argument).

b) Why is the `if` statement required on line 3?

Without the base case, it would invoke itself forever (until running out of memory).

6. A method that invokes itself is called **recursive**. What two steps were necessary to define the `factorial` method? How were these steps implemented in Java?

1. The base case, which was implemented using an if statement.
2. The recursive case, which as implemented using a method call.

7. How many distinct method calls would be made to `factorial` to compute the factorial of 3? Identify the value of the parameter n for each of these separate calls.

Four method calls: `factorial(3)` $\rightarrow$ `factorial(2)` $\rightarrow$ `factorial(1)` $\rightarrow$ `factorial(0)`.

8. Here is the complete output from the program in #5. Identify which distinct method call printed each line. In other words, which lines were printed by `factorial(3)`, which lines were printed by `factorial(2)`, and so on.

`n is 3`	`factorial(3)`
`need factorial of 2`	`factorial(3)`
`n is 2`	`factorial(2)`
`need factorial of 1`	`factorial(2)`
`n is 1`	`factorial(1)`
`need factorial of 0`	`factorial(1)`
`n is 0`	`factorial(0)`
`factorial of 0 is 1`	`factorial(1)`
`factorial of 1 is 1`	`factorial(2)`
`factorial of 2 is 2`	`factorial(3)`
`6`	`main`

9. What happens if you try to calculate the factorial of a negative number? How could you prevent this behavior in the `factorial` method?

The recursion would repeat until the program runs out of memory (`StackOverflowError`). To fix this bug, you could add an if statement that checks for `n < 0` and returns -1.

10. Trivia question: What is the largest factorial you can compute in Java when using `int` as the data type? If you don't know, how could you find out?

$12! = 479,001,600$. Anything larger exceeds the 32-bit range. 20! is the largest for `long` integers. You can find this out by trail and error (e.g., using JShell).

Model 2 Summation

"In mathematics, ***summation*** (capital Greek sigma symbol: Σ) is the addition of a sequence of numbers; the result is their sum or total."

$$\sum_{i=1}^{100} i = 1 + 2 + 3 + \ldots + 100 = 5050$$

Source: https://en.wikipedia.org/wiki/Summation

Questions (20 min) Start time:

11. Consider how to calculate $\sum_{i=1}^{4} i = 10$.

a) Write out all the numbers that need to be added:

$\sum_{i=1}^{4} i =$ $4 + 3 + 2 + 1$

b) Show how this sum can be calculated in terms of a smaller summation.

$\sum_{i=1}^{4} i =$ $4 + \sum_{i=1}^{3} i$

12. Write an expression similar to #11b showing how any summation of n integers can be calculated in terms of a smaller summation.

$$\sum_{i=1}^{n} i = \mathrm{n} + \sum_{i=1}^{n-1} i$$

13. What is the base case of the summation? (Write the complete formula, not just the value.)

$$\sum_{i=1}^{1} i = 1$$

14. Implement a recursive method that takes a single parameter `n` and returns the sum $1 + 2 + \ldots + n$. It should only have an `if` statement and two `return` statements.

```
public static int summation(int n) {

    if (n == 1) {
        return 1;
    } else {
        return n + summation(n - 1);
    }

}
```

15. Discuss how the `factorial` method below uses temporary variables. What lines would you have to change to implement the `summation` method instead?

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;  // base case
    }
    int recurse = factorial(n - 1);
    int result = n * recurse;
    return result;
}
```

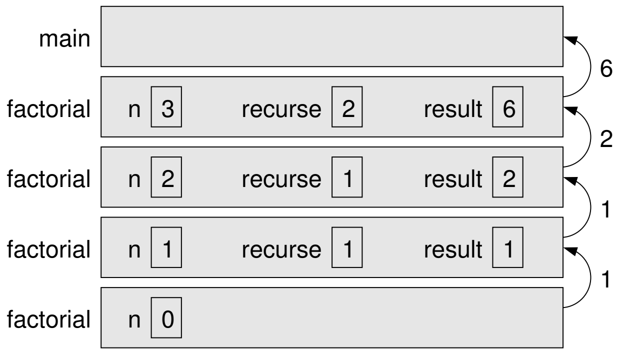
1) Rename the method to `summation`.

2) Change the base case to be `if (n == 1)`.

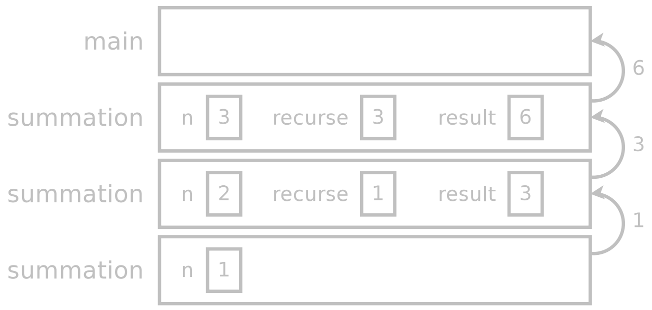
3) The recursive step must invoke `summation`.

4) The result must add instead of multiply.

16. Here is a stack diagram of `factorial(3)` when invoked from `main`. Draw a similar diagram for `summation(3)` as described in the previous question.



main
factorial n 3 recurse 2 result 6
factorial n 2 recurse 1 result 2
factorial n 1 recurse 1 result 1
factorial n 0
6
2
1
1



main
summation n 3 recurse 3 result 6
summation n 2 recurse 1 result 3
summation n 1
6
3
1

17. Why are there no values for `recurse` and `result` in the stack diagram for the last call to `factorial` (when `n == 0`)?

The method returns without declaring and using those variables.

18. Looking at the stack diagram, how is it possible that the parameter `n` can have multiple values in memory at the same time?

Each distinct method call has its own memory for parameters and local variables. The value of `n - 1` in the first method call becomes the value of `n` in the next.