- Warmup: In IntelliJ, fill in the fact function that takes a single int argument (num) and returns the product of all the integers between 1 and num.
  - Use a for loop.
  - Test in main when you're done.
- (This is actually a useful function in science and mathematics, called the factorial function.)
- Compare with your neighbor to see if you did it the same way.



- Warmup: In IntelliJ, fill in the fact function that takes a single int argument (num) and returns the product of all the integers between 1 and num.
  - Use a for loop.
  - Test in main when you're done.
- (This is actually a useful function in science and mathematics, called the factorial function.)
- Compare with your neighbor to see if you did it the same way.

```
public static long fact(int num) {
  long answer = 1;
 for (int i = 1; i <= n; i++) {
    answer *= x;
  return answer;
```

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = 1 \* 2 \* 3 \* 4 \* 5
- Notice that each product involves computing the entire product on the row above.

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = 1 \* 2 \* 3 \* 4 \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = 1 \* 2 \* 3 \* 4 \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = 1 \* 2 \* 3 \* 4 \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = 1 \* 2 \* 3 \* 4 \* 5
- Let's reformulate the definition of a factorial to take advantage of this.

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = 1 \* 2 \* 3 \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = 1 \* 2 \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = fact(2) \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = 1 \* 2
- fact(3) = fact(2) \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = fact(1) \* 2
- fact(3) = fact(2) \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = fact(1) \* 2
- fact(3) = fact(2) \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5

- Let's look at this problem a different way:
- fact(1) = 1
- fact(2) = fact(1) \* 2
- fact(3) = fact(2) \* 3
- fact(4) = fact(3) \* 4
- fact(5) = fact(4) \* 5
- Notice how for n >= 2, each factorial is defined in terms of a smaller factorial.
- So if n >= 2, what is fact(n)?
  - fact(n) = fact(n-1) \* n

#### Recursion

 A recursive function is a function that calls itself.

 Recursive functions are used to solve problems where the solution to the problem involves solving one or more smaller versions of the same problem.

- A recursive function has two parts:
- Base case: How to solve the smallest version(s) of the problem that we care about.
- Recursive case: How to reduce a bigger version of the problem to one or more smaller versions.
  - In order to work, the recursive case (when applied over and over) must eventually reduce every size of the problem down to the base case.
- What are these for factorial?
- Let's write this in Java.

## Thinking Recursively

```
if (problem is sufficiently simple) {
   Directly solve the problem.
   Return the solution.
else {
   Split the problem up into one or more smaller
      problems with a similar structure as the
      original.
   Solve each of those smaller problems.
   Combine the results to get the overall solution.
   Return the overall solution.
```

#### How does this work in Java?

- Recursion works (in all modern programming languages) because:
  - All variables are local.
  - We get new memory for local variables every time a function is called.
- Let's look at a memory diagram when we call factRec(3).

## Why is this useful?

- Any loop (for/while) can be replaced with a recursive function that does the same thing.
  - Some languages don't include loops!
- Because we started with Python and Java, we naturally see things in terms of loops.
- Some problems have a "naturally" recursive solution that is hard to solve with a loop.
- Other problems have solutions that work equally well *recursively* or with loops (*iteratively*).

# Demo

# How to "get" recursion

- Forget all loops.
- To find the base case:

- An "instance" of a problem is a single example or occurrence of that problem.
- "What is the smallest version of this problem I would ever care about solving?"
- To find the recursive case:
  - "If I have an *instance* of the problem, how can I phrase how to solve the problem in terms of solving one or more smaller instances?"

#### **Trust** the recursion

- Base case is usually easy ("When do I stop?")
- In recursive case:
  - Break the problem into multiple parts (not necessarily the same size):
    - A small part I can solve "now."
    - The answer(s) from smaller instance(s) of the problem.
  - Assume the recursive call does the right thing.
  - Figure out how to combine the multiple parts.