Lesson 24: Recursive Algorithms #1 (W07D3)

Balboa High School

Michael Ferraro

October 2, 2015

Do Now

```
public static int mysteryFcn(int n) {
    //precondition:  n > 0

    int result = 1;

    while ( n >= 1 ) {
        result *= n;
        n--;
    }

    return result;
}
```

Consider the code above for the mystery function above, `mysteryFcn()`.

1. What does `mysteryFcn(3)` return? Make a table!
2. How about `mysteryFcn(5)`?
3. What's a more fitting name for `mysteryFcn()`?

Aim

Students will begin working with recursive algorithms, learning their two key attributes and tracing the execution of examples.

Demonstration (1 of 2)

Watch as I write a program to…

- evaluate an infinite sum involving alternating $\pm$ terms
  - $\pm$ via
    - `if()/else`
    - powers of -1
  - infinite loops via
    - `1==1`
    - `true`

Demonstration (2 of 2)

Watch as I write a program to...

- compute an estimate of π using Euler's[1] infinite sum:

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \dots$$

  - solve for π
  - `Scanner` to ask for # of iterations

[1] Leonhard Euler was a famous Swiss mathematician. See here.

Check-In Re: PS #4a

- Due date: **Monday, 5 October 2015, before 5th Period**
- Extra help over coming days:
  - Room 319 during lunch and study hall
  - Room 124 when afterschool help announced
- Today's material is related to PS #4b
- Where are you getting stuck in §3.5 (book problems)?

What is *Recursion*?

- Recursion, in programming, happens when a procedure[2] accomplishes a task by calling upon itself.
- Some problems lend themselves to being solved in such a way, while others are more easily solved using iterative[3] algorithms.
- Any method that can be written recursively can be written iteratively — though doing so may not be trivial!

[2] Think *method* in Java.

[3] Think `while()` loops.

Example of Recursion: Factorial

- Recall the definition of *factorial*:
  $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

Example of Recursion: Factorial

- Recall the definition of *factorial*:
  $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$
- Ex: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Example of Recursion: Factorial

- Recall the definition of *factorial*:
  $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$
- Ex: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- You can write an iterative procedure to figure out $n!$:

```
public static int factorialIter(int n) {
    int result = 1;

    while ( n >= 1 ) {
        result *= n;
        n--;
    }

    return result;
}
```

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times 4!$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times (4 \times 3!)$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times (4 \times (3 \times 2!))$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
    - $5! = 5 \times (4 \times (3 \times (2 \times 1!)))$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
    - $5! = 5 \times (4 \times (3 \times (2 \times (1))))$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times (4 \times (3 \times (2)))$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
    - $5! = 5 \times (4 \times (6))$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times (24)$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
    - $5! = 120$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times 4!$
  - in general terms:
    $n! = n \times (n-1)!$

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times 4!$
  - in general terms:
    $n! = n \times (n-1)!$
- Let's watch this single-stepping video of a **broken** recursive factorial function.
- Save this Scratch program file to your Desktop.
- Start cloud-based Scratch and upload the downloaded program file.
- Fix the problem so the final value of 5! is 120.

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times 4!$
  - in general terms:
    $n! = n \times (n - 1)!$
- Consider this *recursive* implementation of a factorial procedure in Java:

```
public static int factorialRec(int n) {
    return n * factorialRec(n-1);
}
```

Example of Recursion: Factorial

- But you can think of *factorial* in terms of itself:
  - $5! = 5 \times 4!$
  - in general terms:
    $n! = n \times (n-1)!$
- Consider this *recursive* implementation of a factorial procedure in Java:

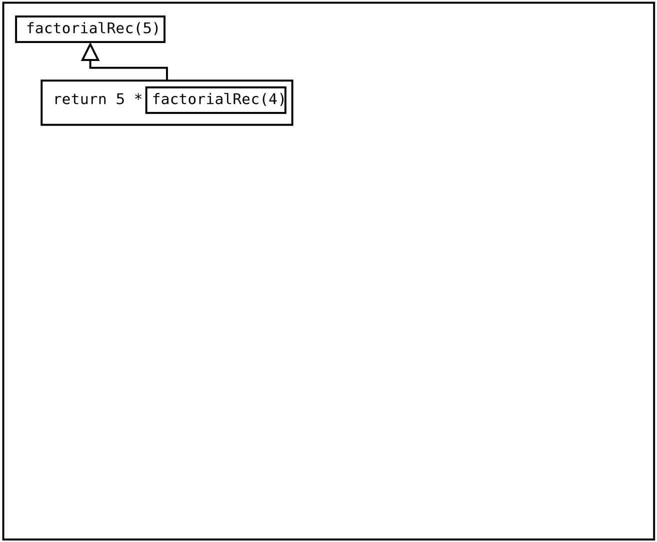
```
public static int factorialRec(int n) {
    return n * factorialRec(n-1);
}
```

- Download `RecursiveFactorial.java` from here, import into a new project called `Lesson24`, and run it.

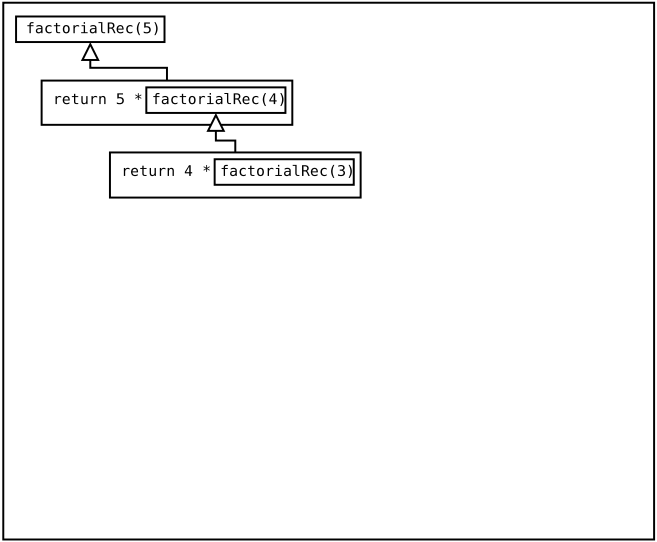
  → **What happens? Why?**

How `factorialRec()` Works Now

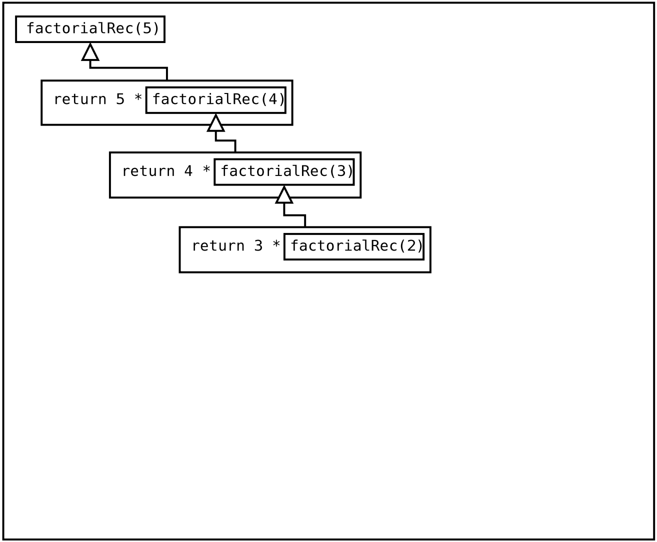
How factorialRec() Works Now



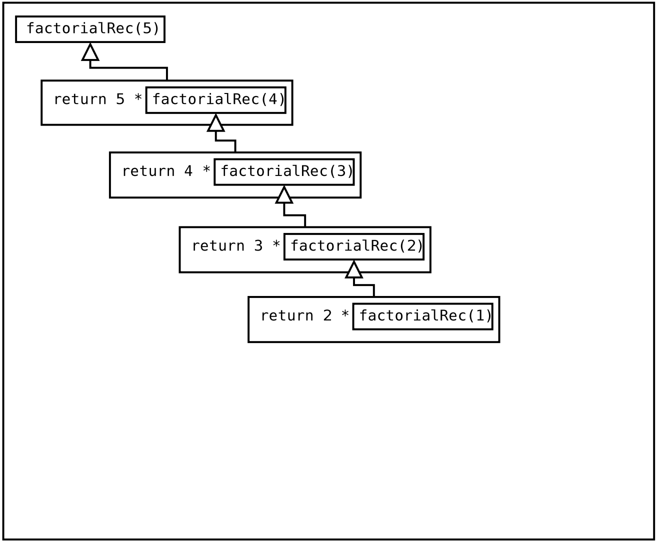
How factorialRec() Works Now



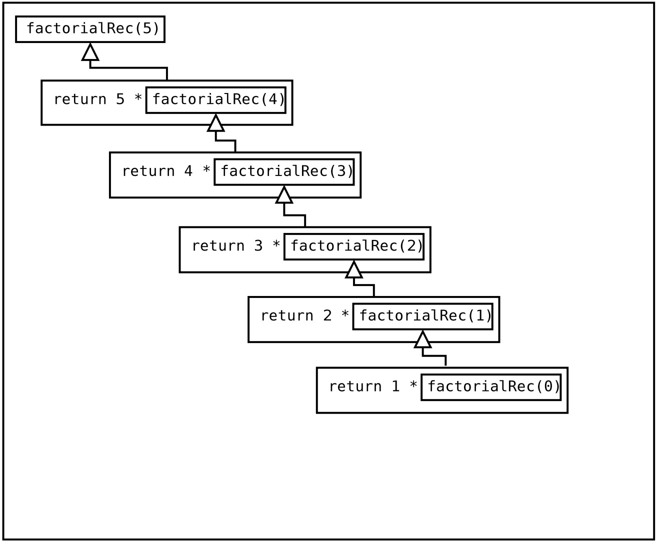
How factorialRec() Works Now



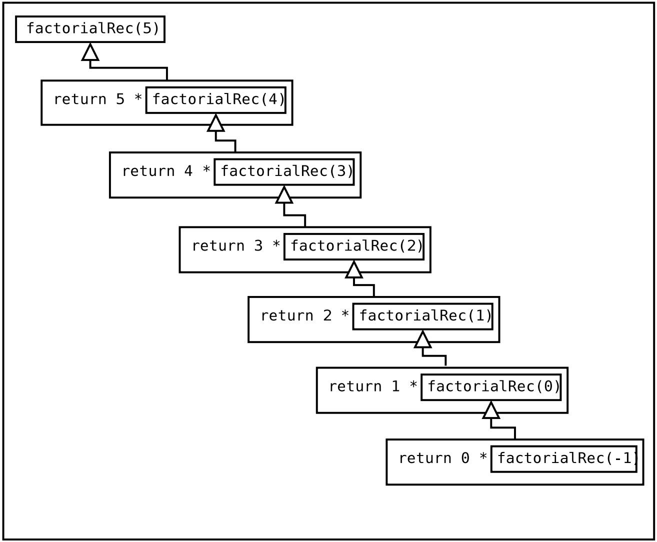
How `factorialRec()` Works Now



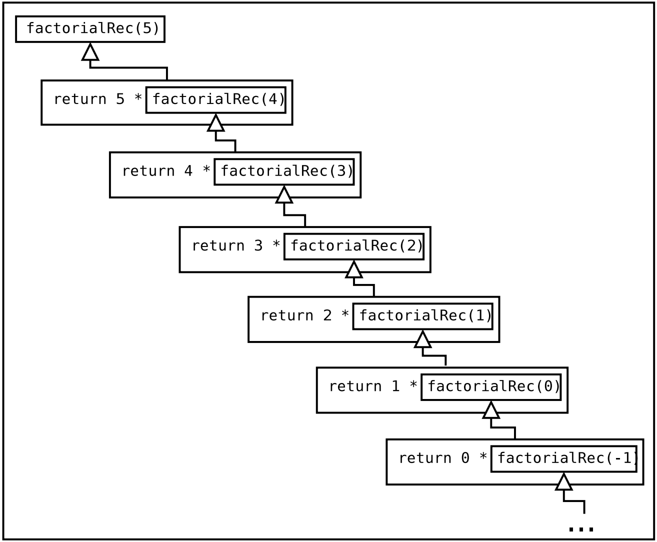
How factorialRec() Works Now



How factorialRec() Works Now



How factorialRec() Works Now



The Two Elements of Recursion

All recursive procedures. . .

1. Make calls to themselves

   Ex: `return n * factorialRec(n-1);`

The Two Elements of Recursion

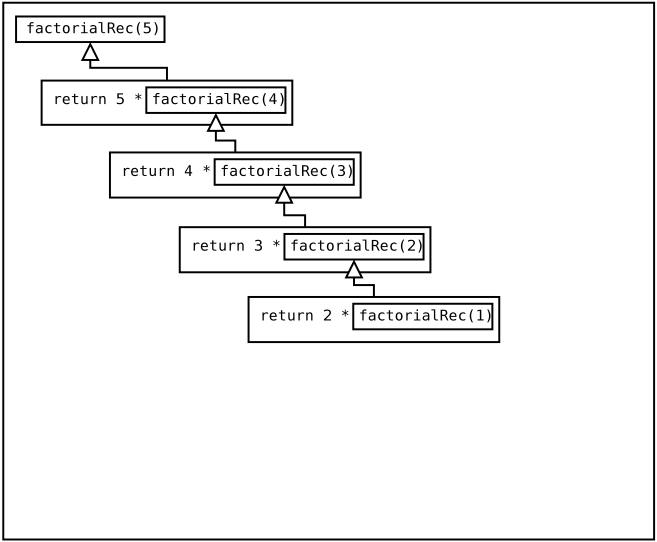
All recursive procedures. . .

1. Make calls to themselves

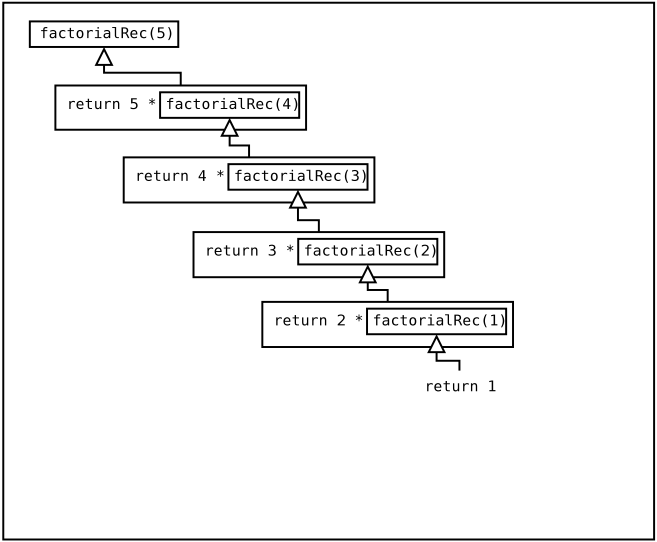
   Ex: `return n * factorialRec(n-1);`

2. Have a *base case*, which acts to stop the recursion
   → **this was missing from `factorialRec.java`!**

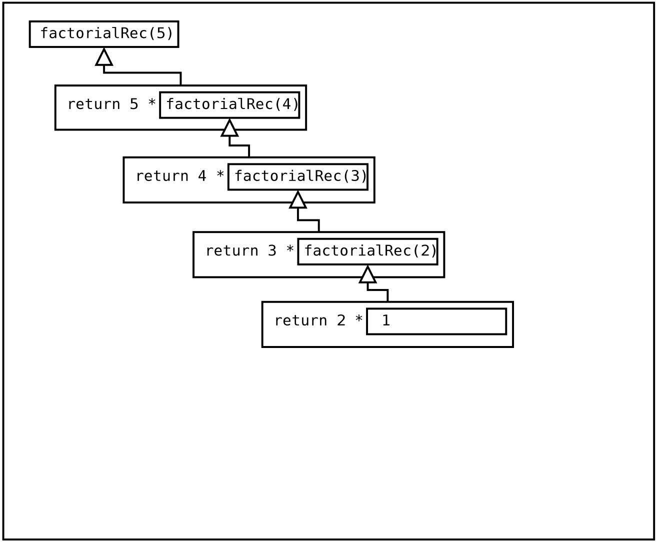
How factorialRec() Should Work



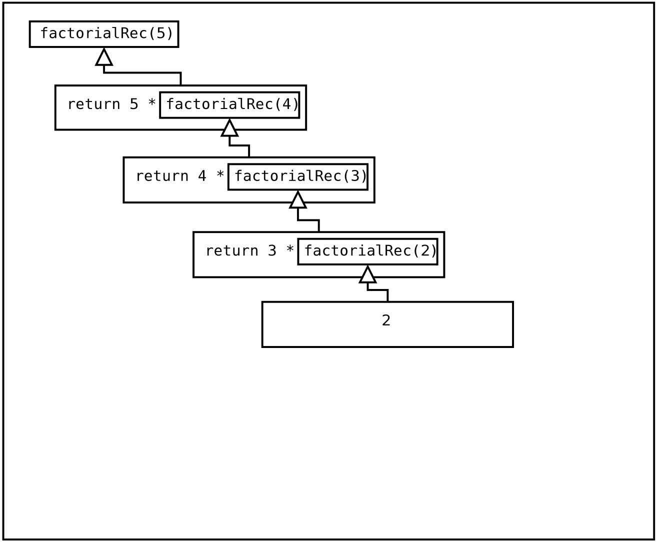
How factorialRec() Should Work



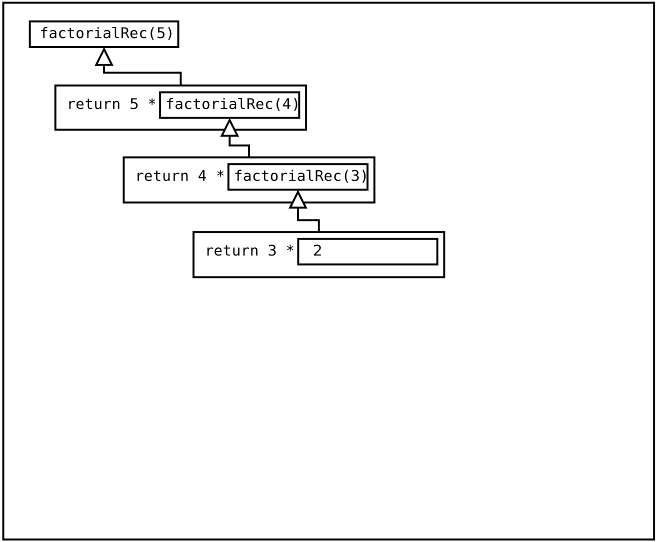
How factorialRec() Should Work



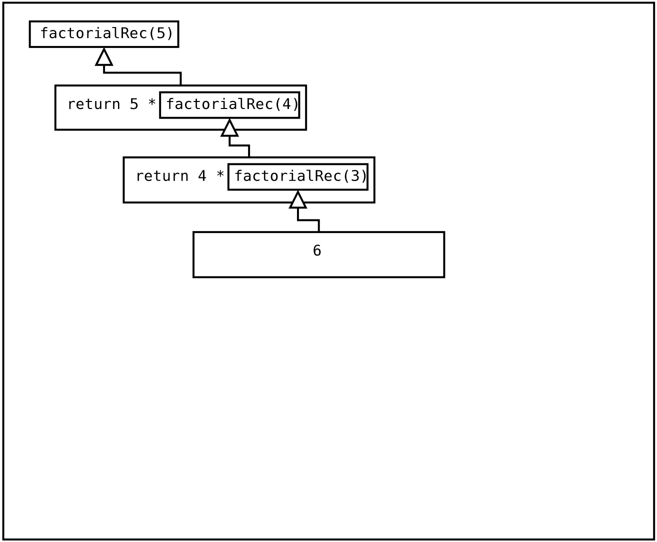
How `factorialRec()` Should Work



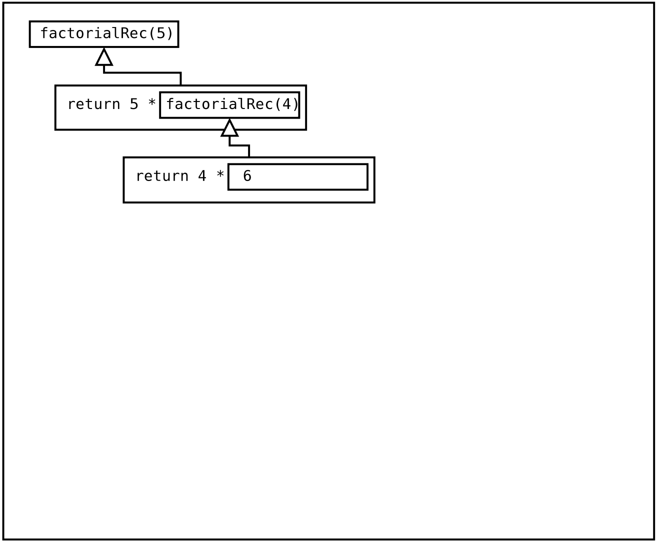
How factorialRec() Should Work



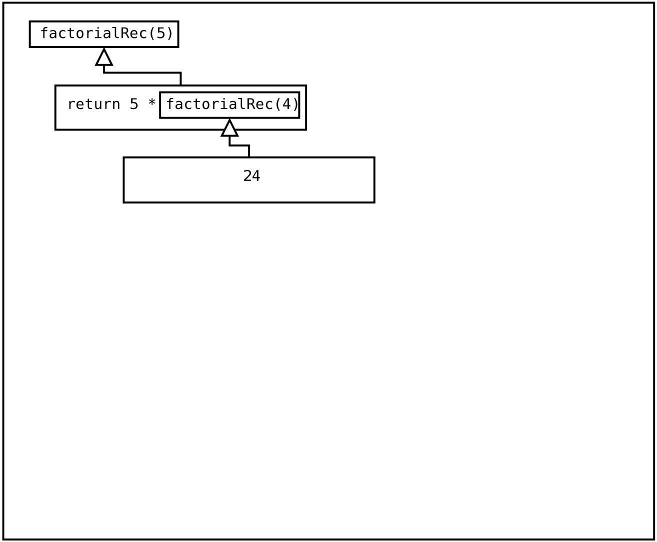
How factorialRec() Should Work



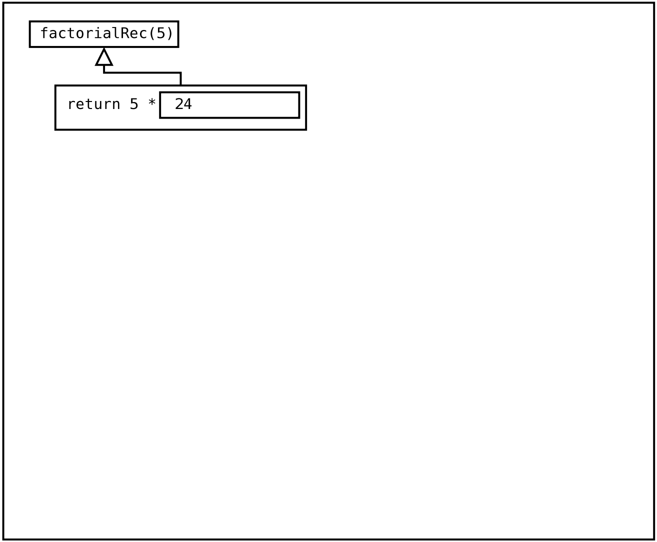
How `factorialRec()` Should Work



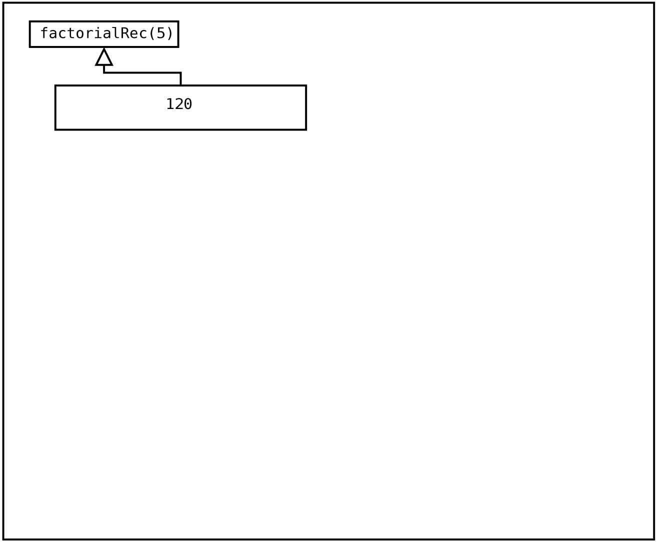
How factorialRec() Should Work



How `factorialRec()` Should Work



How factorialRec() Should Work



Determine the Base Case

- Whenever `factorialRec()` is called, check the value of n:
  - if $n = 1$, simply return 1
  - otherwise, return as we did before

Determine the Base Case

- Whenever `factorialRec()` is called, check the value of n:
  - if $n = 1$, simply return 1
  - otherwise, return as we did before
- Update `factorialRec()`:

```
public static int factorialRec(int n) {
    if ( n == 1 ) {
        return 1;
    }
    return n * factorialRec(n - 1);
}
```

A Simple Example

Using a piece of paper, trace the execution of the code below and state its output.

```
public static int simpleRecEx(int a) {
    if ( a == 0 ) {
        return 0;
    }
    return 2 + simpleRecEx(a - 1);
}

public static void main(String[] args) {
    int result = simpleRecEx(4);
    System.out.println("result is " + result);
}
```

Sum of Squares, Part II

- How could you think of `findSumOfSquares()` in recursive terms?

`findSumOfSquares(n)` $= 1^2 + 2^2 + ... + n^2$

Sum of Squares, Part II

- How could you think of `findSumOfSquares()` in recursive terms?

  `findSumOfSquares(n)` $= 1^2 + 2^2 + ... + n^2$

- First, consider running the sum in reverse:

  `findSumOfSquares(n)` $= n^2 + (n-1)^2 + ... + 1^2$

Sum of Squares, Part II

- How could you think of `findSumOfSquares()` in recursive terms?

  `findSumOfSquares(n)` $= 1^2 + 2^2 + ... + n^2$

- First, consider running the sum in reverse:

  `findSumOfSquares(n)` $= n^2 + (n-1)^2 + ... + 1^2$

- Rewrite the problem in terms of itself:

  `findSumOfSquares(n)` $= n^2 +$ `findSumOfSquares(n-1)`

Sum of Squares, Part II

- How could you think of `findSumOfSquares()` in recursive terms?

  `findSumOfSquares(n)` $= 1^2 + 2^2 + ... + n^2$

- First, consider running the sum in reverse:

  `findSumOfSquares(n)` $= n^2 + (n-1)^2 + ... + 1^2$

- Rewrite the problem in terms of itself:

  `findSumOfSquares(n)` $= n^2 +$ `findSumOfSquares(n-1)`

- Base case: What's the least value of n for which we should compute/evaluate `findSumOfSquares(n)`?

Sum of Squares, Part II

- How could you think of `findSumOfSquares()` in recursive terms?

  `findSumOfSquares(n)` $= 1^2 + 2^2 + ... + n^2$

- First, consider running the sum in reverse:

  `findSumOfSquares(n)` $= n^2 + (n-1)^2 + ... + 1^2$

- Rewrite the problem in terms of itself:

  `findSumOfSquares(n)` $= n^2 +$ `findSumOfSquares(n-1)`

- Base case: What's the least value of *n* for which we should compute/evaluate `findSumOfSquares(n)`?

- Next class: You'll work in pairs to write a recursive version of `findSumOfSquares()`.

HW

Work on completing PS #4a