CSCI 136:
Data Structures
and
Advanced Programming

Lecture 8

Recursion, part 2

Instructor: Dan Barowy

Williams

Announcements

- Lab 3: check email for partner

Outline

1. Recursion
2. Recursion activity
3. Recursion tradeoffs

Recursion

Recursion

- General problem solving strategy
  - Split **big problem** into **smaller sub-problems**.
  - Sub-problems may look a lot like original; are often **smaller versions of same problem**!

Recursion

Recursion is when a thing is **defined in terms of itself**. The most concrete application of recursion in computer science is **when a function is called within its own definition**.

Recursion

```
int countCoins(Tardis jar) {
  if (!empty(jar)) {
    Coin c = takeCoin(jar)
    int total = countCoins(jar)
    if (isUSACoin(c)) {
      return total + c.value;
    } else {
      return total;
    }
  }
  return 0;
}
```

Whenever you "call" `countCoins`, pass the jar to the next person.

Whenever you `return`, write the return value on the slip of paper and pass it to the person that "called" you.

Example: Fibonacci Number

The **nth Fibonacci number** in the **Fibonacci sequence** is defined as:

`0` if `n == 0`,
`1` if `n == 1`, and
`Fibonacci(n-1) + Fibonacci(n-2)` otherwise,

for all `n >= 0`.

Recursion

```
public static int fibonacci(int n){
  if (n == 0){
    return 0;
  }
  if (n == 1){
    return 1;
  }

  return fibonacci(n - 1) +
         fibonacci(n - 2);
}
```

Recursion: formal structure

- Recursion is a good solution when a problem fits a **basic pattern**:
- It has at least one "terminating" rule that **does not** use recursion, called the **base case**.
- It has at least one rule that **does** use recursion, called the **recursive case**. The recursive case should **reduce the problem toward the base case**.

Factorial

- `n! = n × (n-1) × (n-2) × … × 1`
- How can we implement this?
  - We could use a `for` loop…

```
int product = 1;
for(int i = 1;i <= n; i++)
    product *= i;
```

- But we could also write it recursively….

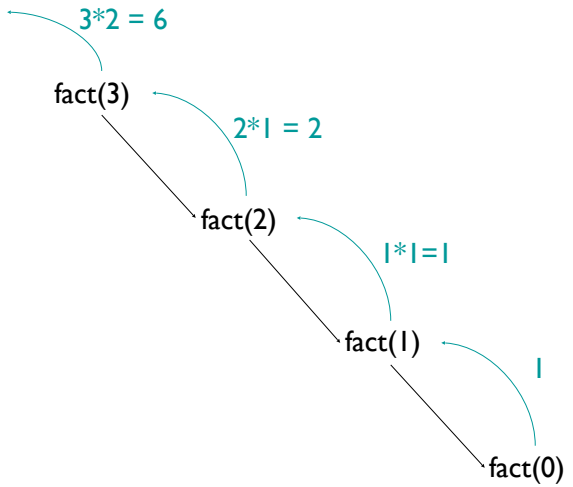
Activity: Factorial

- `n! = n × (n-1) × (n-2) × … × 1`
- Work with a partner and see if you can come up with a recursive solution.

How did we know to look for that insight?

- n! = n × (n-1) × (n-2) × … × 1
- n! = n × (n-1)!
- 0! = 1

Graphically…



Factorial

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

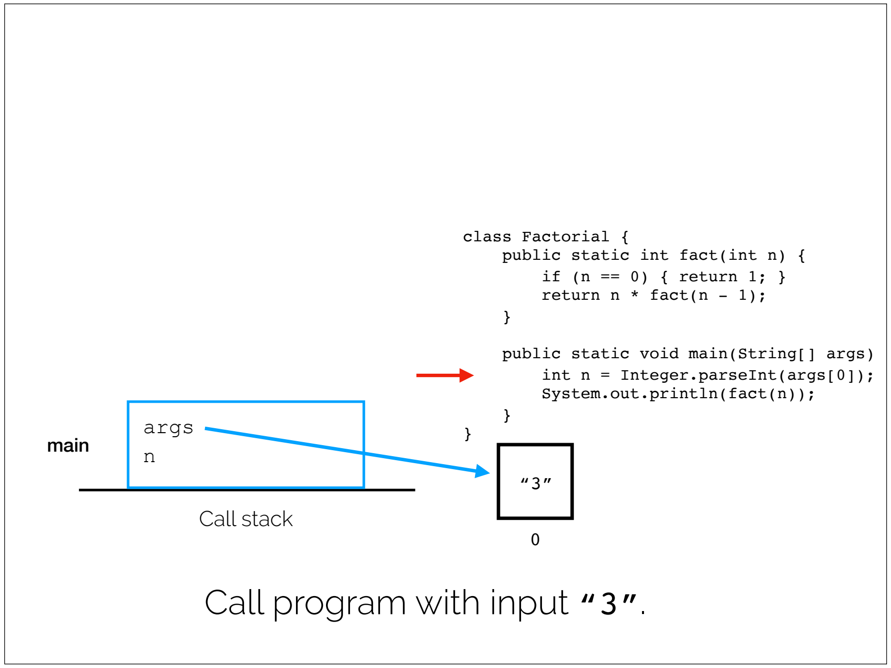
What could I do to improve this?

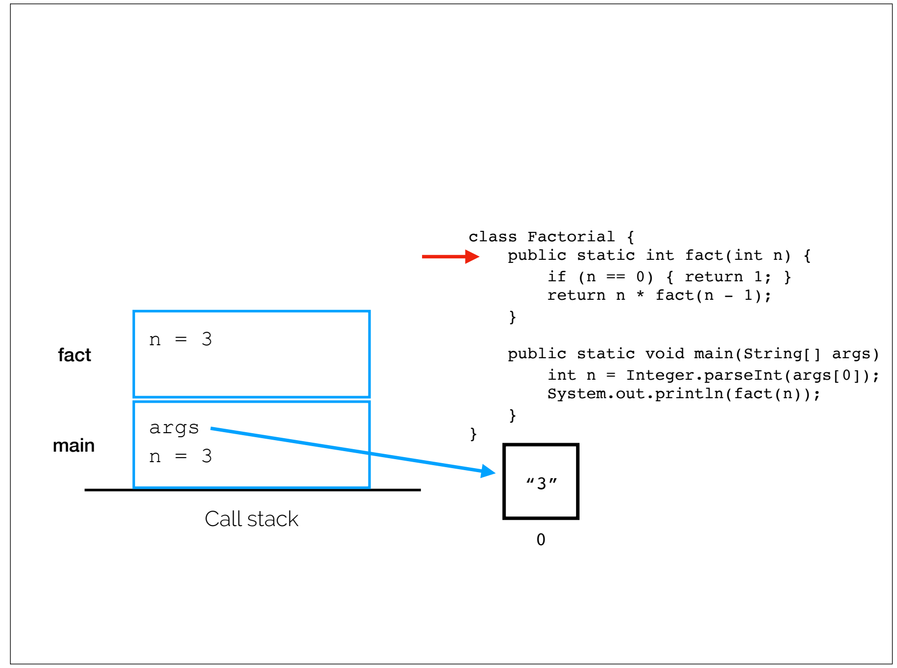
(Hint: what values of **n** are permissible?)

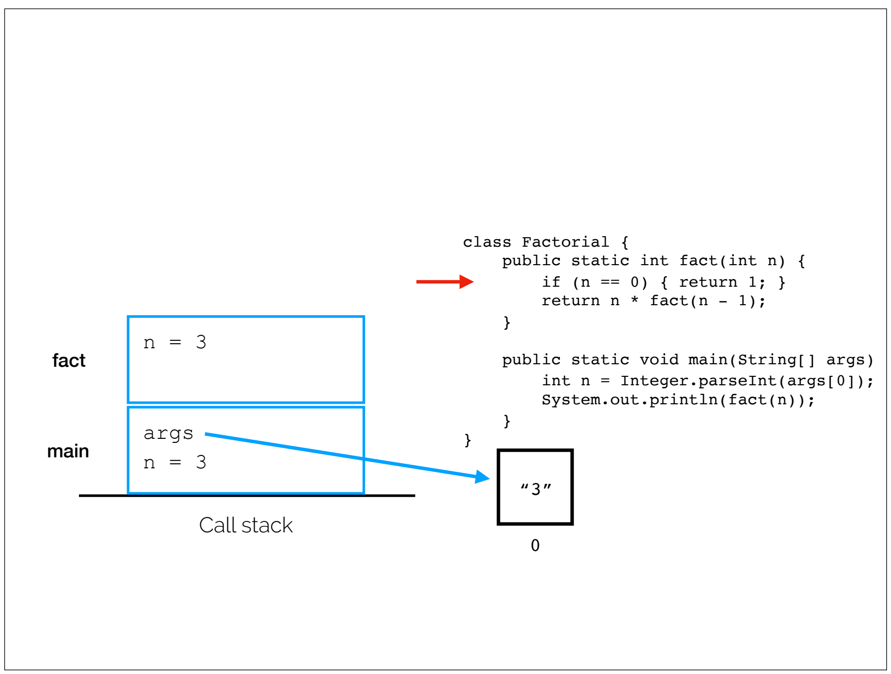
```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

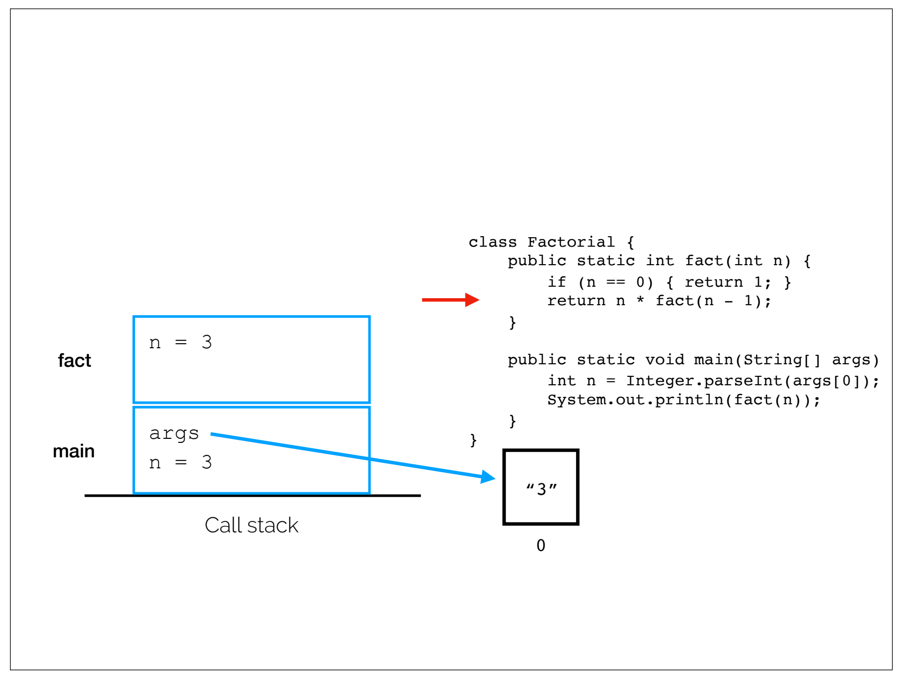
Call stack

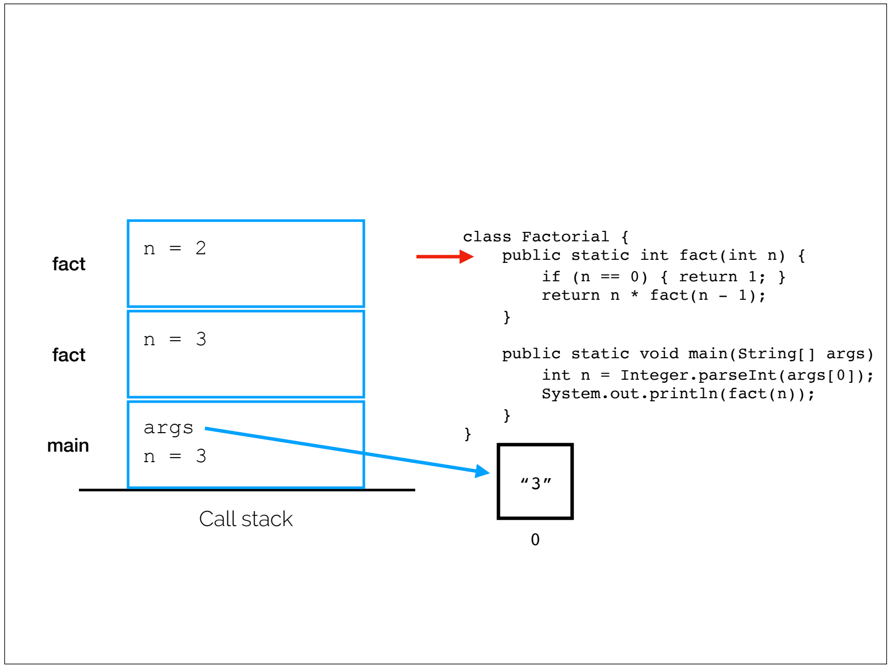
Call program with input **"3"**.

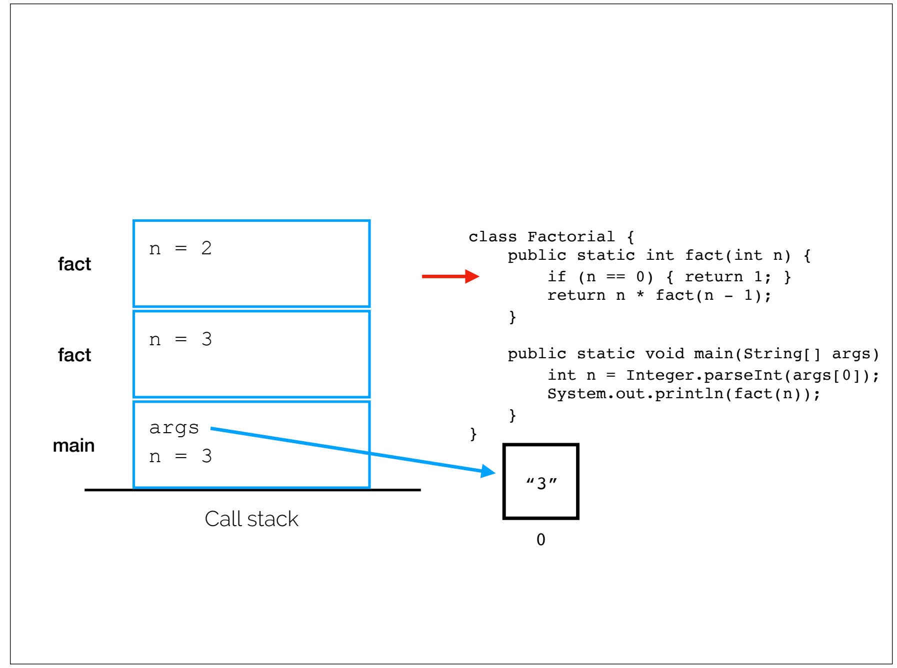

class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
main
args
n
"3"
0
Call stack
Call program with input "3".

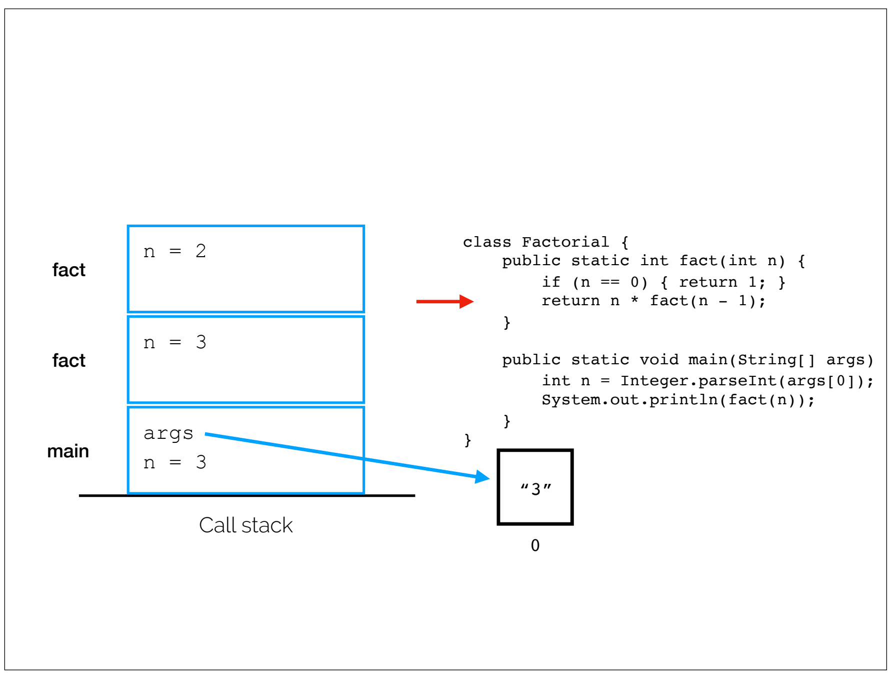

class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
fact
n = 3
main
args
n = 3
"3"
0
Call stack

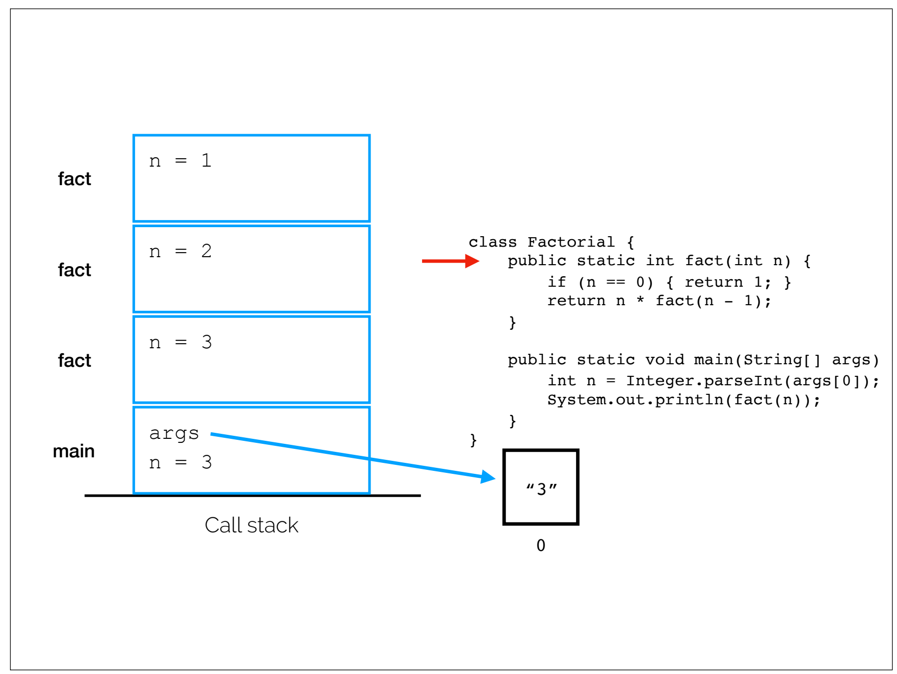

class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
fact
n = 3
main
args
n = 3
"3"
0
Call stack

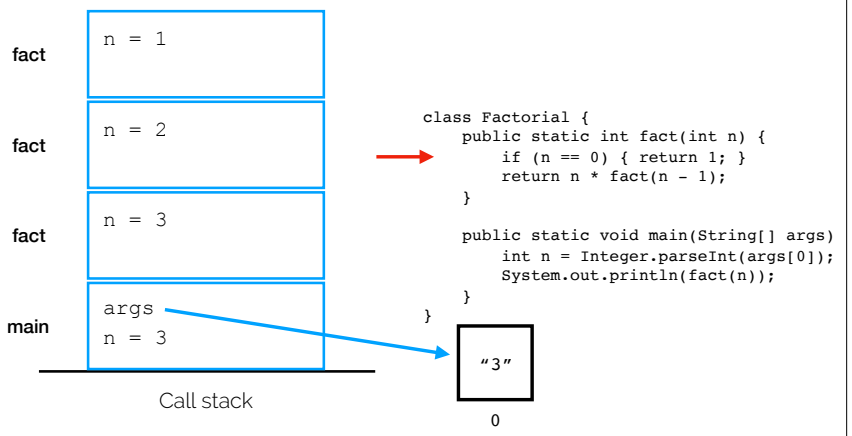

class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
fact
n = 3
main
args
n = 3
"3"
0
Call stack

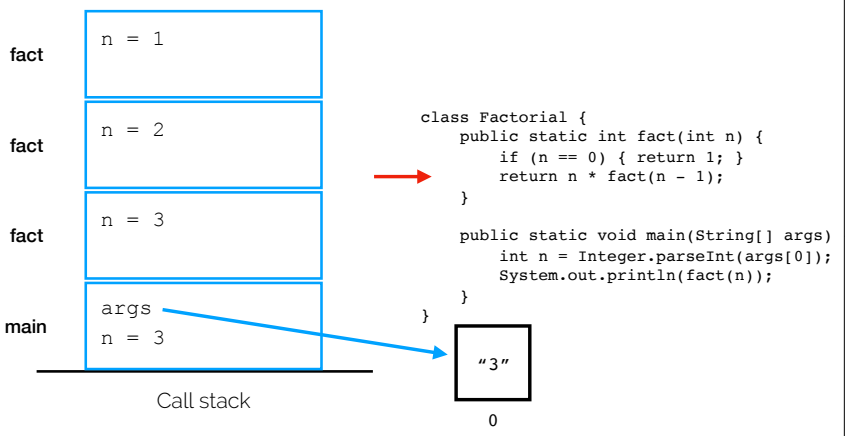

fact
n = 2
fact
n = 3
main
args
n = 3
"3"
0
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

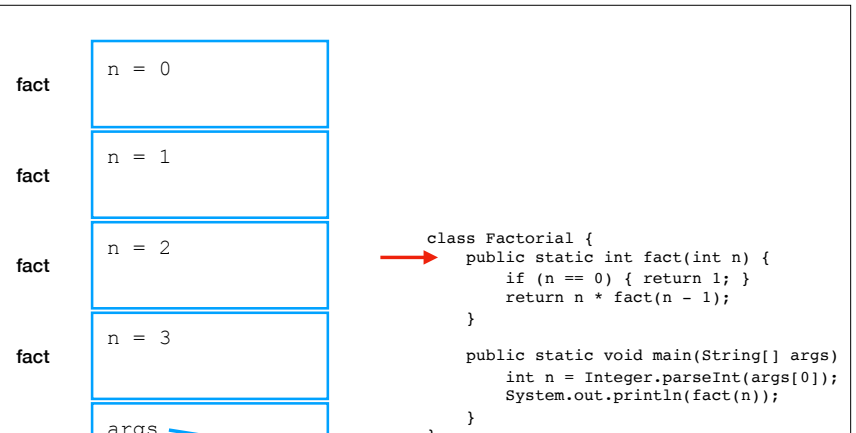

fact
n = 2
fact
n = 3
main
args
n = 3
"3"
0
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

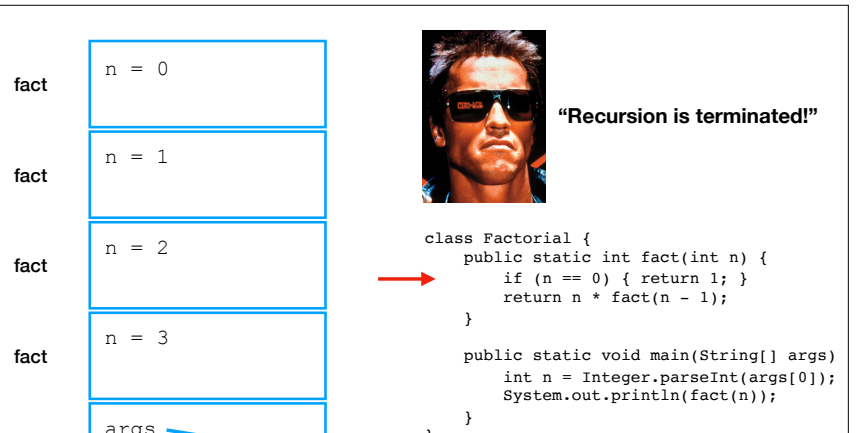

fact
n = 2
fact
n = 3
main
args
n = 3
"3"
0
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}


fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
"3"
0
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

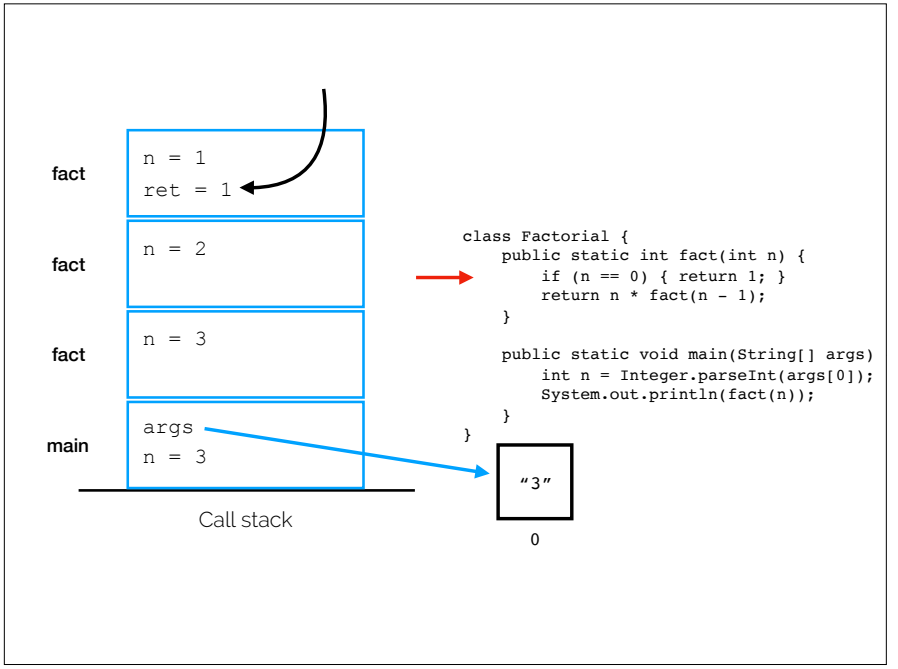

fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0

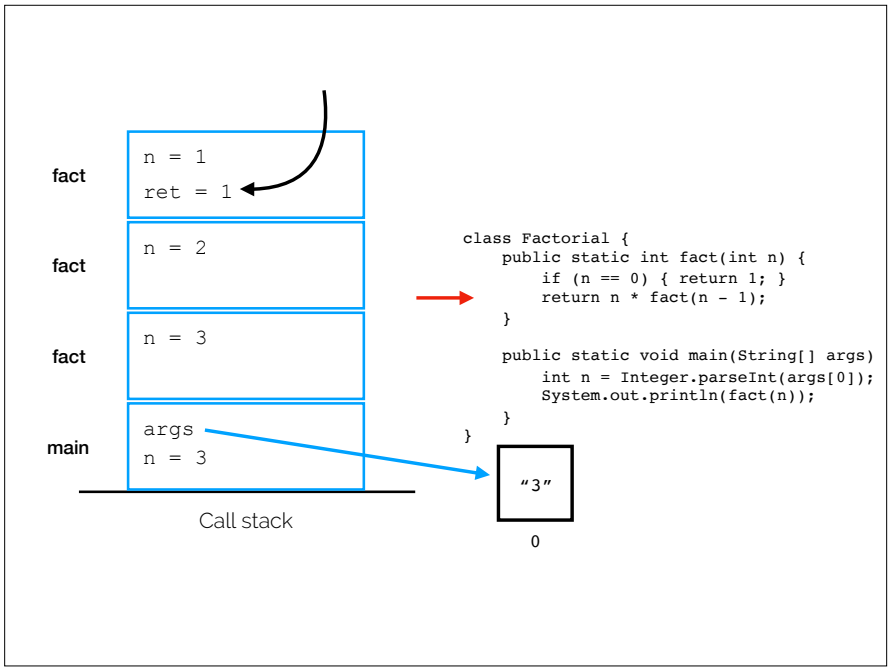

fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0

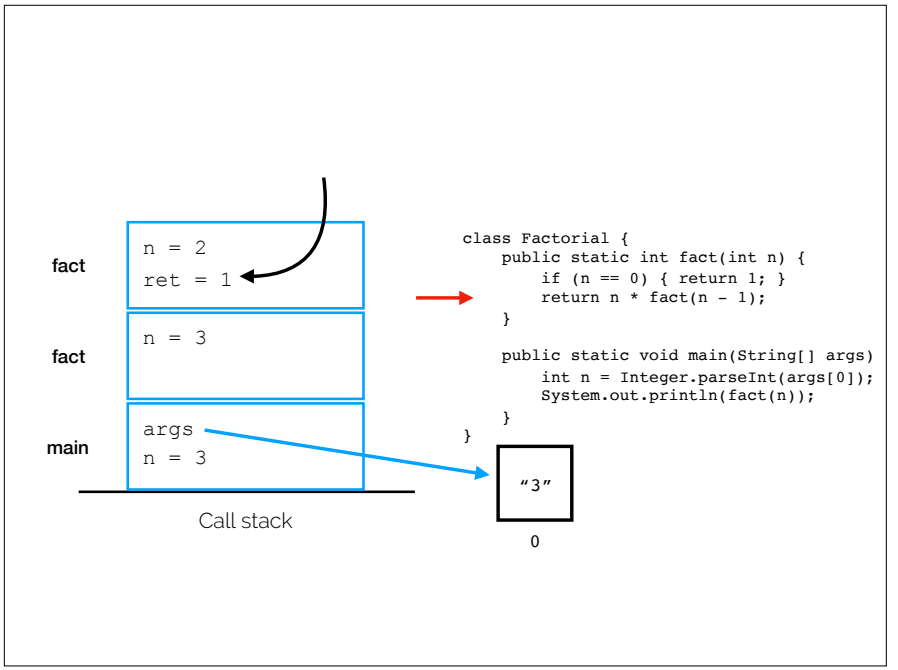

fact
n = 0
fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0

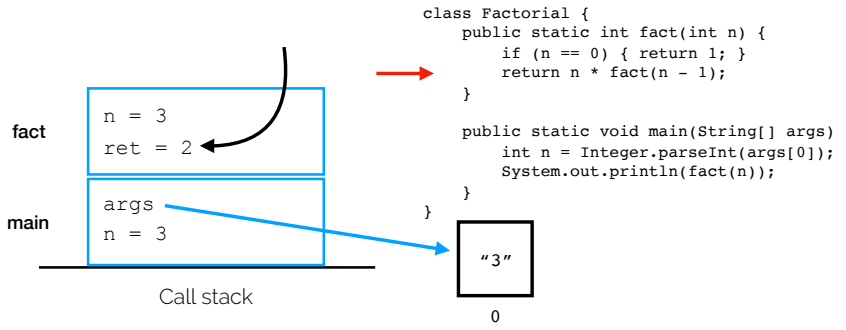

fact
n = 0
fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0
"Recursion is terminated!"

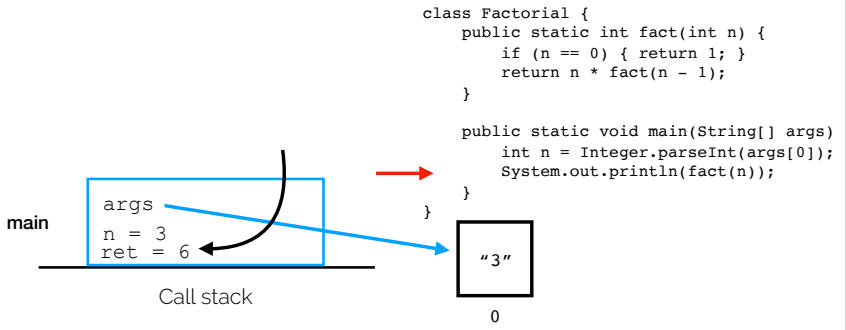

SCHWARZENEGGER
IT'S NOTHING PERSONAL
TERMINATOR 2
JUDGMENT DAY

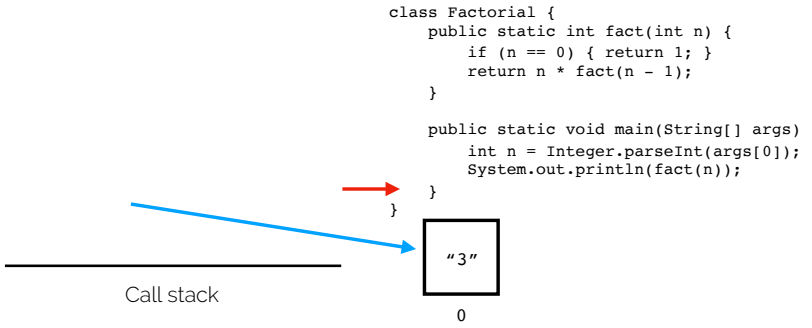

fact
n = 1
ret = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0


fact
n = 1
ret = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0


fact
n = 2
ret = 1
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0







Recursion tradeoffs

- Advantages
  - Often easier to construct recursive solution
  - Code is usually cleaner
  - Some problems do not have obvious non-recursive solutions
- Disadvantages
  - Time cost of recursive calls
  - Memory cost (need to store state for each recursive call until base case is reached)

Linked Lists

Linked List

A **linked list** is a recursive data structure. A linked list is composed of simple pieces called **list nodes**. A list node contains **data** (of generic type **T**) and a **reference** (a "link") to either **another list node** or **null**.

Linked List

(Note: lab 4's linked list is slightly different for reasons that will become clear in that lab.)

∅

The empty list is defined as **null**.

Linked List

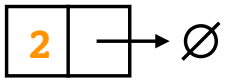
Every other list has at least one list node.

Linked List

A list node stores data of type `T`.

Here, `T` is `Integer`.

Linked List



The `next` field stores a reference ("link") to the next node.

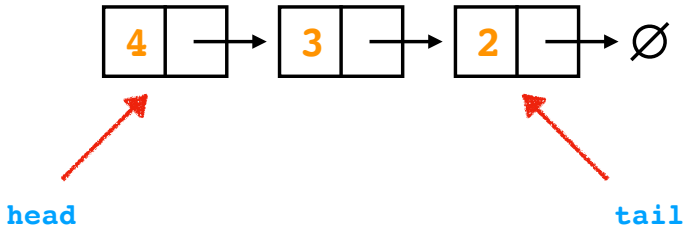
If the node is the last node, the next node is **null**.

Linked List

If the next node is not **null**, it is, recursively, a list node.

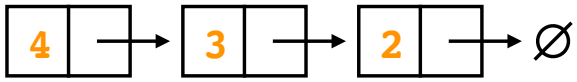
The last node in the list must always point to **null**.

Linked List



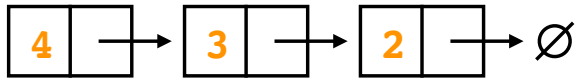
A list has parts.

Linked List



When we add data to a list, we always **append** to the **head**.

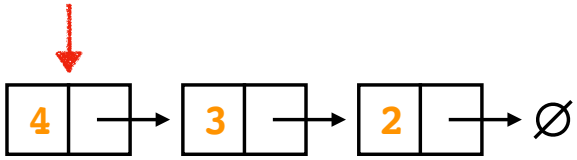
Linked List



To find a value, we must always traverse the list.

E.g., looking for **2**...

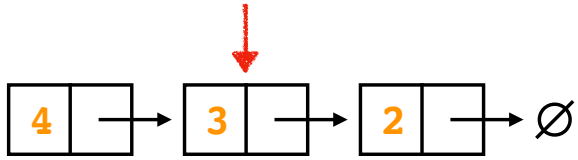
Linked List



To find a value, we must always traverse the list.

E.g., looking for **2**...

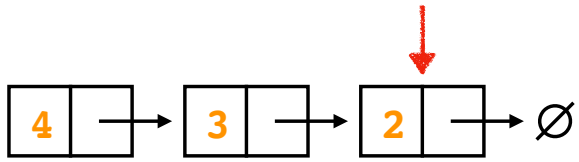
Linked List



To find a value, we must always traverse the list.

E.g., looking for **2**...

Linked List



To find a value, we must always traverse the list.

E.g., looking for **2**...

List API

Recap & Next Class

Today we learned:

- More Recursion
- Recursion activity
- Recursion tradeoffs
- Linked lists

Next class:

- Induction