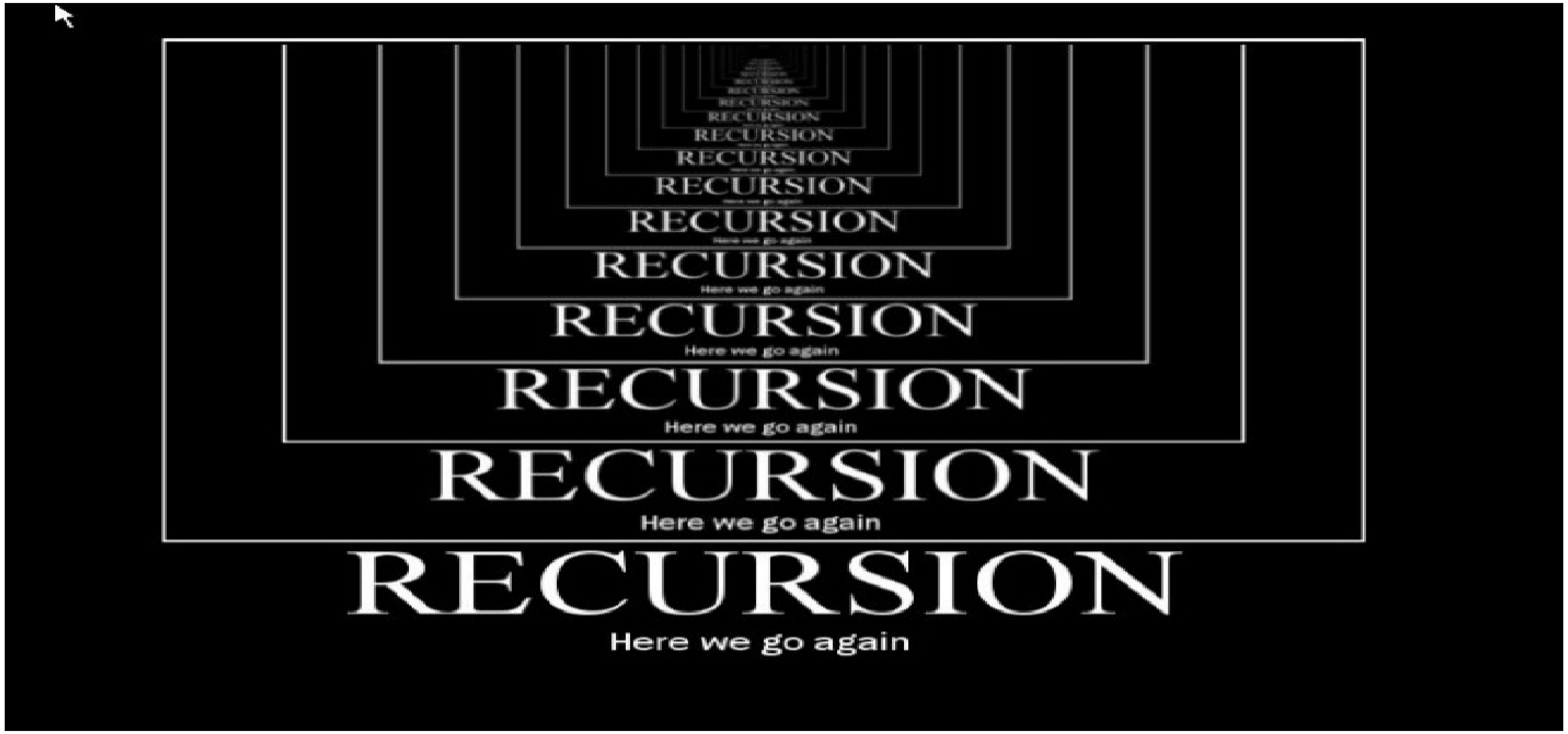
7/20 ICS 141 Programming with Objects

Jessica Maistrovich

Slides provided by Thanaa Ghanem

Metropolitan State University

What is recursion?



Recursive Thinking: An Example

FIGURE 7.1
A Set of Nested Wooden Figures

Strategy for processing nested doll:

```
if you have a present
      enjoy it
else
      open the outer doll
      process the inner nested doll
```

Introduction to **Recursion**

- In programming, the fundamental control structures are:
  - sequence,
  - alternation (`if-else`), and
  - iteration (`while, for`).

- However, sometimes, the best way to solve a problem is by solving a <u>smaller version</u> of the exact same problem first

- Recursion is a technique that solves a problem by solving a <u>smaller problem</u> of the same type

- A recursive method is a method that calls itself. For example, the following **methodX** includes a call to itself

```
public static void methodX(int n){
        System.out.println(n);
        if (n >= 1)
                methodX(n-1);
}
```

Why Recursion?

- Recursion is used to perform repetition in your code (some how similar to loops).

- There are many problems whose solution is defined recursively. For example, the factorial of a positive number **n** can be computed using the factorial of **n-1:**
    - n! = n * (n-1)*(n-2)*.........*1
    - **n!** =
        - If n=1 1
        - If n > 1 **n * (n-1)!**

- Any recursive method can be written using loops, however, for some problems, recursive methods are:
    - much easier to write, and
    - more intuitive

Example 1: Computing 2^n

$$2^n = \begin{cases} 1 & \text{if } n = 0 \\ 2*2^{(n-1)} & \text{if } n > 0 \end{cases}$$

Example using Definition

$2^4 = 2 * 2^3$
$= 2 * 2 * 2^2$
$= 2 * 2 * 2 * 2^1$
$= 2 * 2* \ 2 * 2 * 2^0$
$= 2 * 2 * 2 * 2 * 1$

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```

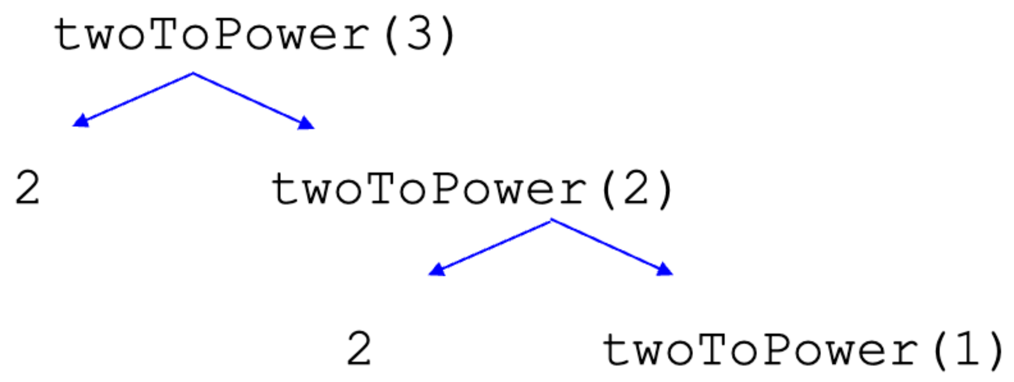
Execution Trace (decomposition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```

```
      twoToPower(3)
       /        \
      2      twoToPower(2)
```

Execution Trace (decomposition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```



Execution Trace (decomposition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```

```
twoToPower(3)
2    twoToPower(2)
     2    twoToPower(1)
          2    twoToPower(0)
```

Execution Trace (composition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```

twoToPower(3)

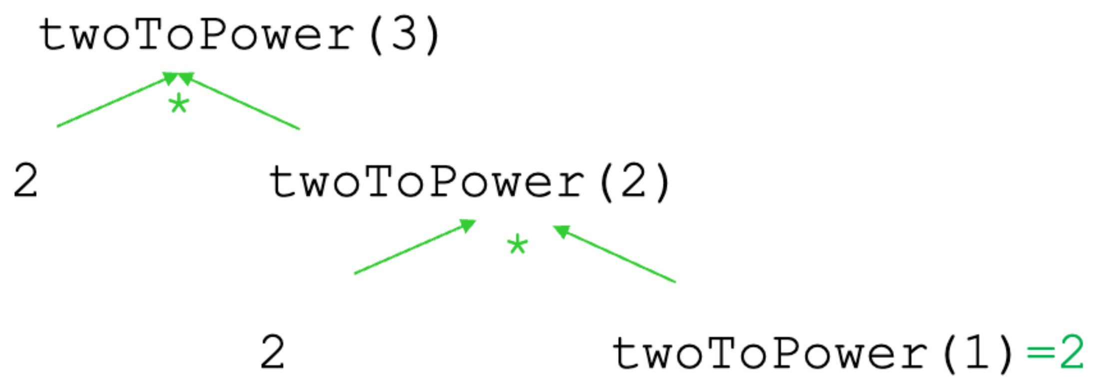
2 * twoToPower(2)

2 * twoToPower(2)

2 twoToPower(0) = 1

Execution Trace (composition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```



Execution Trace (composition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```

twoToPower(3)

*

2 twoToPower(2)=4

Execution Trace (composition)

```
int twoToPower (int n) {
  if (n == 0)
    return 1;
  else
    return 2 * twoToPower(n-1);
}
```

`twoToPower(3)= 8`

Lab part 1

General Form of Recursive Methods

- The over-all strategy is to find some easily applied action that breaks the problem into smaller pieces.
- If the problem is easy, solve it immediately.
- If the problem can't be solved immediately, divide it into smaller problems, then solve the smaller problems using the same method.
- The over-all strategy is to find some easily applied action that breaks the problem into smaller pieces. Some of the pieces can be dealt with immediately; others may need to be further broken up.

General Form of Recursive Methods (continued)

```
solve(problem) {

  if (problem is minimal/not decomposable: a base case)
          solve problem directly; i.e., without recursion
  else {
          1. decompose problem into one or more similar,  strictly smaller sub-
          problems: SP1, SP2, ... , SPN

          2.  recursively call solve on each sub-problem (i.e., solve(SP1),
          solve(SP2),..., solve(SPN))

          3.  combine the solutions to these sub-problems into a solution that
          solves the original problem
  }
}
```

Example for illustration: Fund raising – recursive vs. iterative

Problem: Collect $1,000.00 for charity

Assumption: Everyone is willing to donate a penny

- **Iterative Solution**
  - Visit 100,000 people, asking each for a penny

- **Recursive Solution**
  - If you are asked to collect only one penny, give a penny to the person who asked you for it
  - Otherwise
    - Visit 10 people and ask them to each raise 1/10th of the amount of money that you have been asked to raise
    - Collect the money that they give you and combine it into one bag
    - Give it to the person who asked you to collect the money

Example 2: Triangular Number

- Formula for Triangle number:

```
Triangular( N ) = N + Triangular( N-1 )
```

- The number of pins in 1 row is called a **base case**.

- A base case is a problem that can be solved immediately.

Triangle Numbers		
number	Triangular(number)	Formula
1	1	base case
2	3	2 + Triangle(1)
3	6	3 + Triangle(2)
4	10	4 + Triangle(3)
5	15	5 + Triangle(4)
6	21	6 + Triangle(5)
7	28	7 + Triangle(6)

Triangular Number: From Math to Java

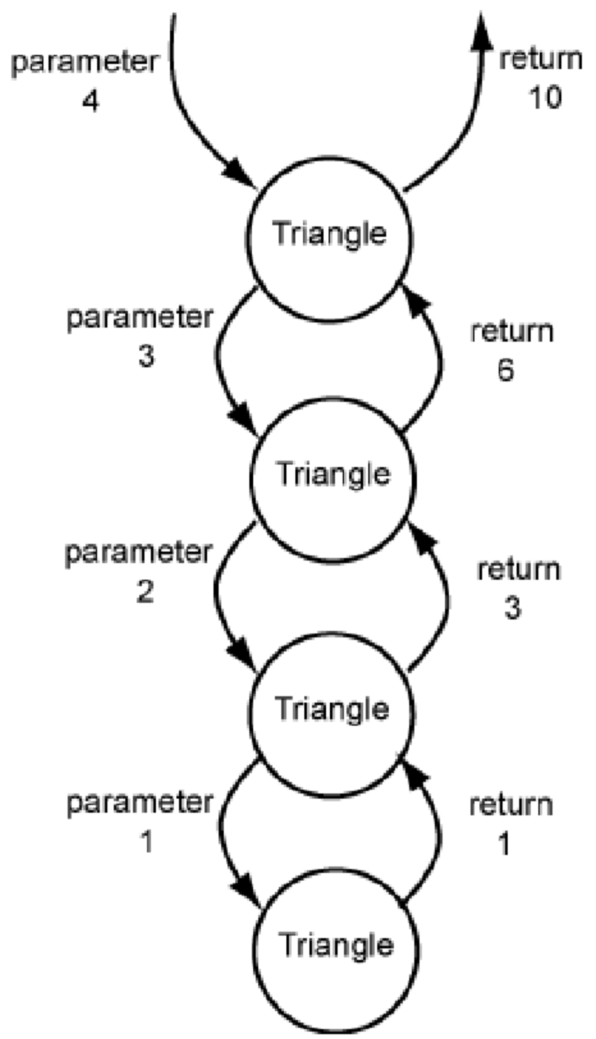
<table>
<tr><th>Definition</th><th>Translation into Java</th></tr>
<tr><td><pre>Triangle(1) = 1

Triangle(N) = N + Triangle(N-1)</pre></td><td><pre>public int triangle(int N) {
        if (N == 1)
              return 1;
       else
              return N + Triangle(N-1);
}</pre></td></tr>
</table>

Executing

`int result = triangle(4)`

http://programmedlessons.org/Java9/chap91/ch91_18.html



Example 3: Factorial

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n*(n-1)! & \text{if } n > 1 \end{cases}$$

Example using Definition
4! = 4 * 3!
= 4 * 3 * 2!
= 4 * 3 * 2 * 1!
= 4 * 3 * 2 * 1 * 0!
= 4 * 3 * 2 * 1 * 1

```
int factorial (int n){
  if (n == 1) //Non-decomposable
    return 1;
  else {
      //decompose
      int subN    = n-1;
      //solve sub-problem
      int subFact = factorial(subN);
      //combine
      return n * subFact;
  }
}
```

can be simplified to

```
int factorial (int n) {
  if (n == 1)
    return 1;
  else
    return n * factorial(n-1);
}
```

```
int factorial (int n) {
  if (n  == 1)
    return 1;
  else
    return n * factorial(n-1);
}
```

return 6

```
int factorial (int 3) {
  if (n == 1)
    return 1;
  else
    return 3 * factorial(2);
}
```

return 2

```
int factorial (int 2) {
  if (n == 1)
    return 1;
  else
    return 2 * factorial(1);
}
```

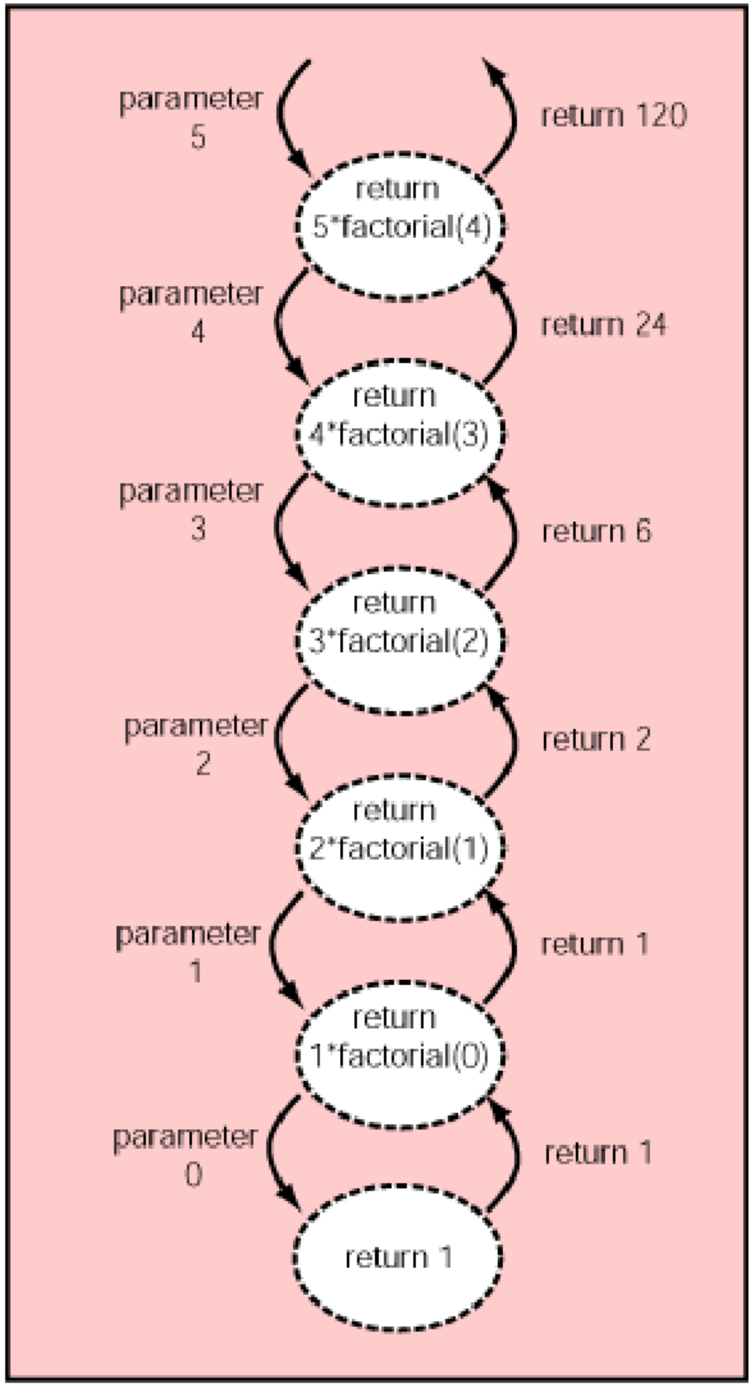
return 1

```
int factorial (int 1) {
  if (n == 1)
    return 1;
  else
    return 2 * factorial(1);
}
```

Executing

`int num = factorial(5)`

- http://programmedlessons.org/Java9/chap92/ch92_04.html



Key Components of a Recursive Algorithm Design

1. What is a smaller ***identical*** problem(s)?
   - Decomposition

2. How are the answers to smaller problems combined to form the answer to the larger problem?
   - Composition

3. Which is the smallest problem that can be solved easily (without further decomposition)?
   - Base/stopping case

Two Things to get right

(1) Be sure to test for each base case.

(2) Be sure to divide the other cases into smaller parts.

- Usually in the Java translation, the base case is detected using an `if`-statement.
  - Sometimes there are several base cases. Be sure to include all of them.

An example on two base cases

- Convert the following Math definition to a recursive method in Java (don't worry about what it defines):

```
Thing( 0 ) = 0

Thing( 1 ) = 1

Thing( N ) = 2*N + Thing( N/2 -1 )
```

```
public int Thing( int N ){
  if ( N == 0 )
    return 0;
  else if ( N == 1 )
    return 1;
  else
    return 2*N + Thing( N/2 - 1 );
}
```

What is the output of the following method when n=3

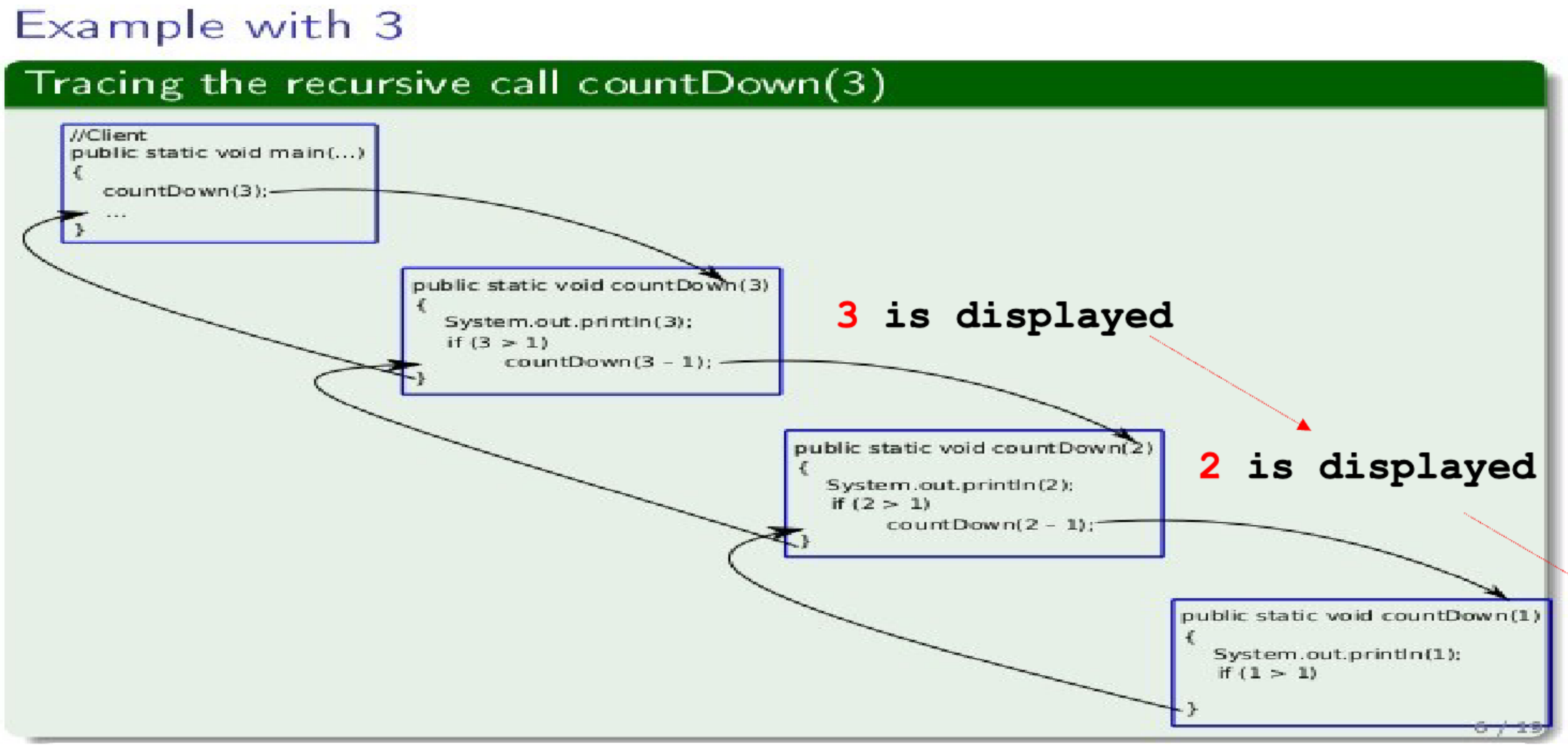
```java
public static void countDown(int n){
        System.out.println(n);
        if (n > 1)
                countDown(n-1);
}
```

Output will be:

```
3
2
1
```

Tracing countDown method

Output will be:
3
2
1



What is the output of the following method when n = 3?

```
public static void f2(int n){
        if (n > 1)
                f2(n-1);
        System.out.println(n);
}
```

Output will be:

```
1
2
3
```

```
public static void main(String[] args ){
        f2(3);
}
```

```
public static void f2(3){
        if (3 > 1)
                f2(2);
        System.out.println(3);
}
```

3 is displayed

```
public static void f2(2){
        if (2 > 1)
                f2(1);
        System.out.println(2);
}
```

2 is displayed

```
public static void f2(1){
        if (1 > 1)
                f2(1);
        System.out.println(1);
}
```

1 is displayed

Output will be:

```
1
2
3
```

Computational Advantage of recursion vs. iteration

- Recursion is useful because sometimes a problem is naturally recursive, then all you need to do is match it with Java code.

- But recursion does not add any fundamental power to Java.
  - Any method that uses recursion can be written using iteration.
  - Any method that uses iteration can be written using recursion.

- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code