SFWR ENG/COMP SCI 2S03 Principles of Programming

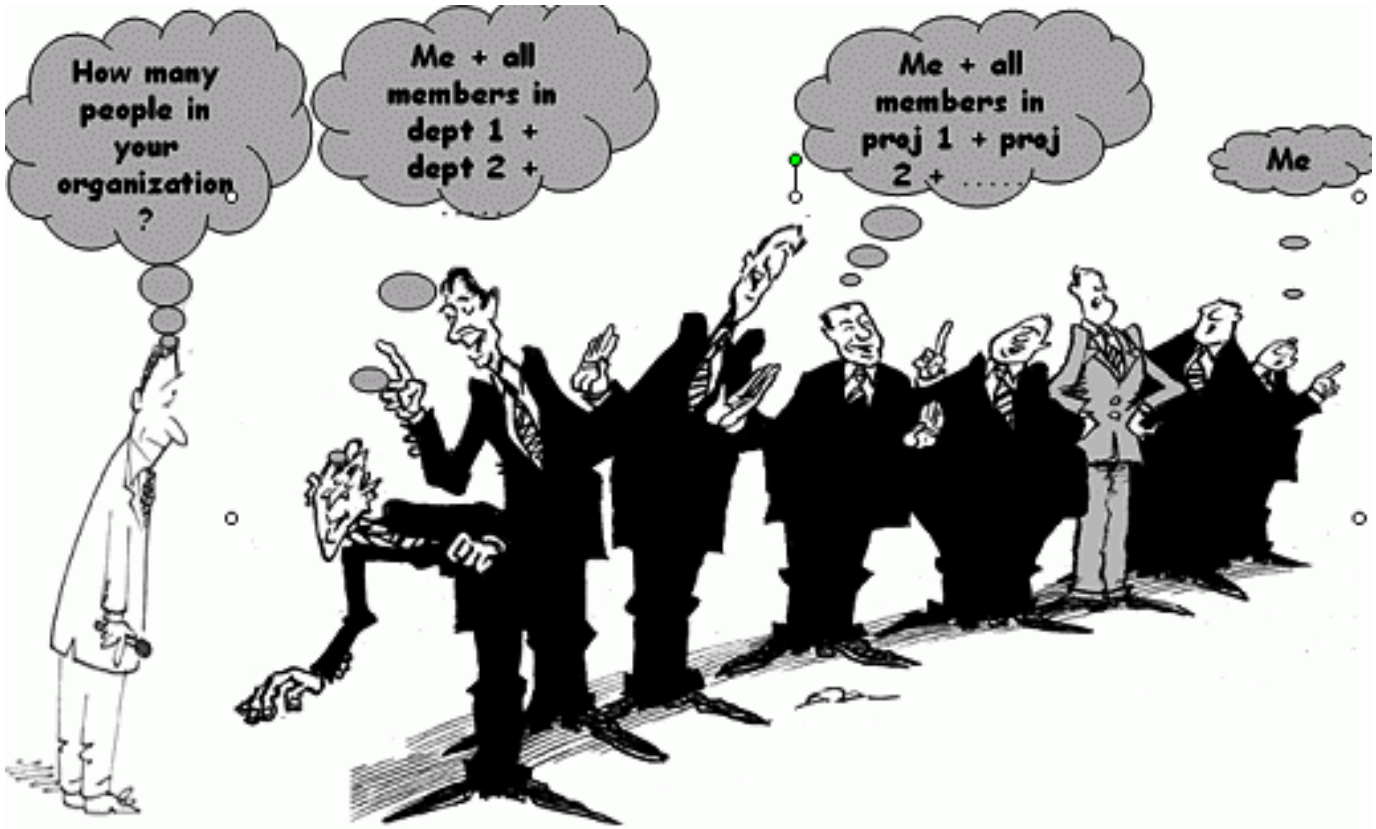
Dr. Ridha Khedri

Department of Computing and Software, McMaster University
Canada L8S 4L7, Hamilton, Ontario

Acknowledgments: Material based on *Java actually: A Comprehensive Primer in Programming* (Chapter 17)

Topics Covered

Introduction and Learning Objectives



Introduction and Learning Objectives

- It is usually prefered to define a function directly in terms of variables
- It is sometimes convenient (or even necessary) to use a method called recursion
- Principle of induction:
  - A function is defined for a given starting value a (usually 0 or 1), and
  - If, when it is defined for k greater than a, it can then be defined for the value $k + 1$

  then the function can be defined for all integers greater than a.

Axiom (Mathematical Induction over $\mathbb{N}$)

$$P(0) \ \wedge \ (\ \forall(n : \mathbb{N} \ |: \ \forall(i \ | \ 0 \leqslant i \leqslant n \ : \ P(i) \ ) \implies P(n+1) \ )) \implies \forall(n : \mathbb{N} \ |: \ P(n) \ )$$

- Conjunct $P(0)$ is called the base case of the mathematical induction
- $\forall(n : \mathbb{N} \ |: \ \forall(i \ | \ 0 \leqslant i \leqslant n \ : \ P(i) \ ) \implies P(n+1) \ )$ is called the inductive case of the mathematical induction
- Recursive definitions can be used for functions that admit induction

- Induction can be performed over any subset $n_0, n_0 + 1, n_0 + 2, \cdots$, of the integers

- The only difference in such a proof is the starting point and thus the base case

Theorem (Mathematical Induction over $\{n_0, n_0 + 1, \cdots\}$)

$$P(n_0) \ \wedge \ (\ \forall (n : \mathbb{N} \mid n_0 \leqslant n \ : \ \forall (i \mid n_0 \leqslant i \leqslant n \ : \ P(i) \ ) \implies P(n+1) \ ))$$
$$\implies \ \forall (n : \mathbb{N} \mid n_0 \leqslant n \ : \ P(n) \ )$$

Suppose, we want to define b^n for $b : \mathbb{Z}$ and $n : \mathbb{N}$

- $b^n = \cdot(i \mid 1 \leqslant i \leqslant n : b)$
- An alternative style:

$$\begin{cases} b^0 & = & 1 \\ b^{n+1} & = & b \cdot b^n \text{ (for } n \geqslant 0) \end{cases}$$

- Or,

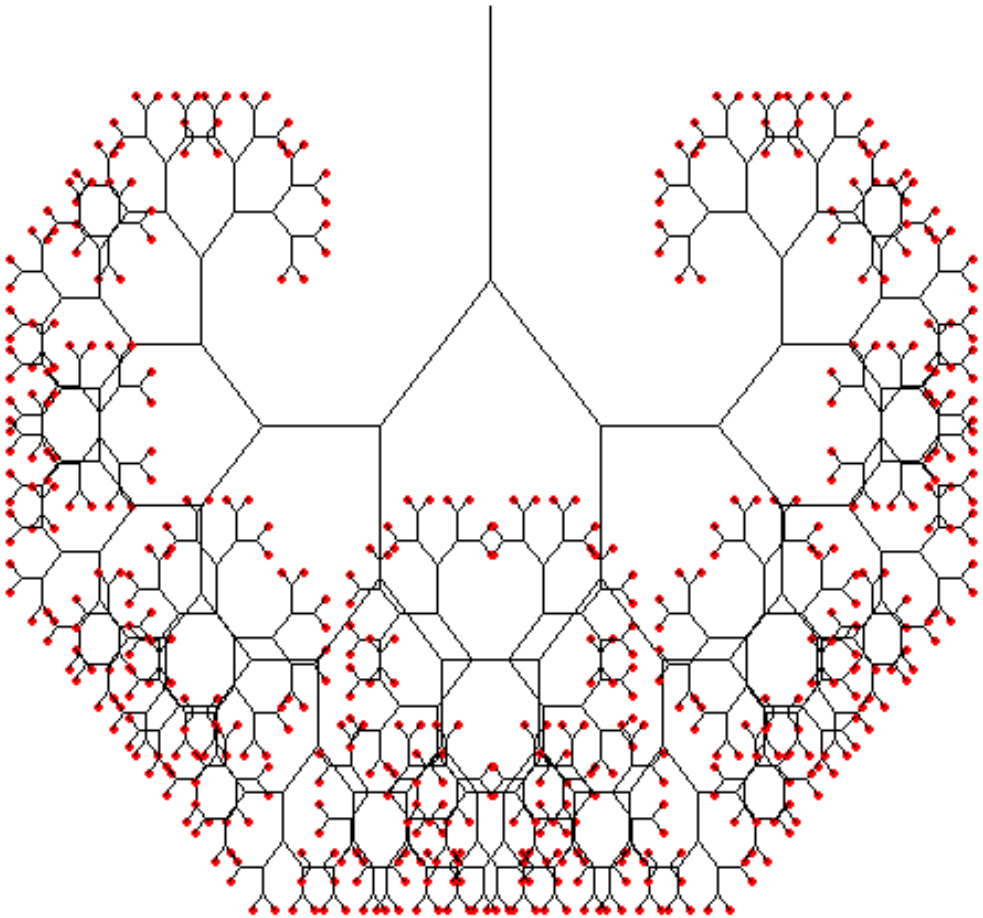
$$\begin{cases} b^0 & = & 1 \\ b^n & = & b \cdot b^{n-1} \text{ (for } n \geqslant 1) \end{cases}$$

Problem

A model for the number of lobsters caught per year is based on the assumption that the number of lobsters caught in a year is the average of the number caught in the two previous years. At the beginning of the application of this model, $100,000$ *lobsters were caught in year 1 and* $300,000$ *were caught in year 2.*
Define inductively L_n*, where* L_n *is the number of lobsters caught in year n, under the assumption of this model and its initial conditions.*

Introduction and Learning Objectives

- Recursion occurs when an operation uses itself as part of its execution
- Recursion is used as a problem-solving technique in which a problem is divided into smaller versions of itself
- + the partial problems leads to the solution of the overall problem
- In Java, recursive programming is implemented by methods that call themselves



Learning Objectives

- What recursion is and how it can be applied
- Why recursive algorithms must always specify base cases
- That a recursive algorithm always has an iterative implementation
- Gain insight into recursive programming by looking at several problems that have recursive solutions
- Potential pitfalls of recursive programming

$$n! \stackrel{\text{def}}{=} \begin{cases} 1! & = & 1 \\ n! & = & n \cdot (n-1)! \text{ (for } n \geqslant 2) \end{cases}$$

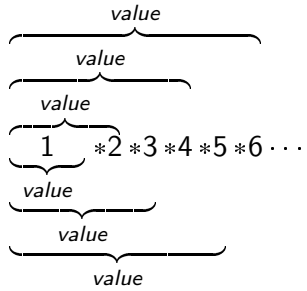
$$\text{factorial}(n) \stackrel{\text{def}}{=} \begin{cases} \text{factorial}(1) & = & 1 \\ \text{factorial}(n) & = & n \cdot \text{factorial}(n-1) \text{ (for } n \geqslant 2) \end{cases}$$

- This is a recursive definition, since it uses itself as part of the definition

```
1  // Iterative calculation of factorial numbers.
   class IterativeFactorial {
3    public static void main(String[] args) {
       System.out.println("5! = " + factorial(5));
5    }

7    static int factorial(int n) {
       int value = 1;
9
       for (int i = 2; i <= n; i++)
11       value *= i;

13     return value;
     }
15 }
```



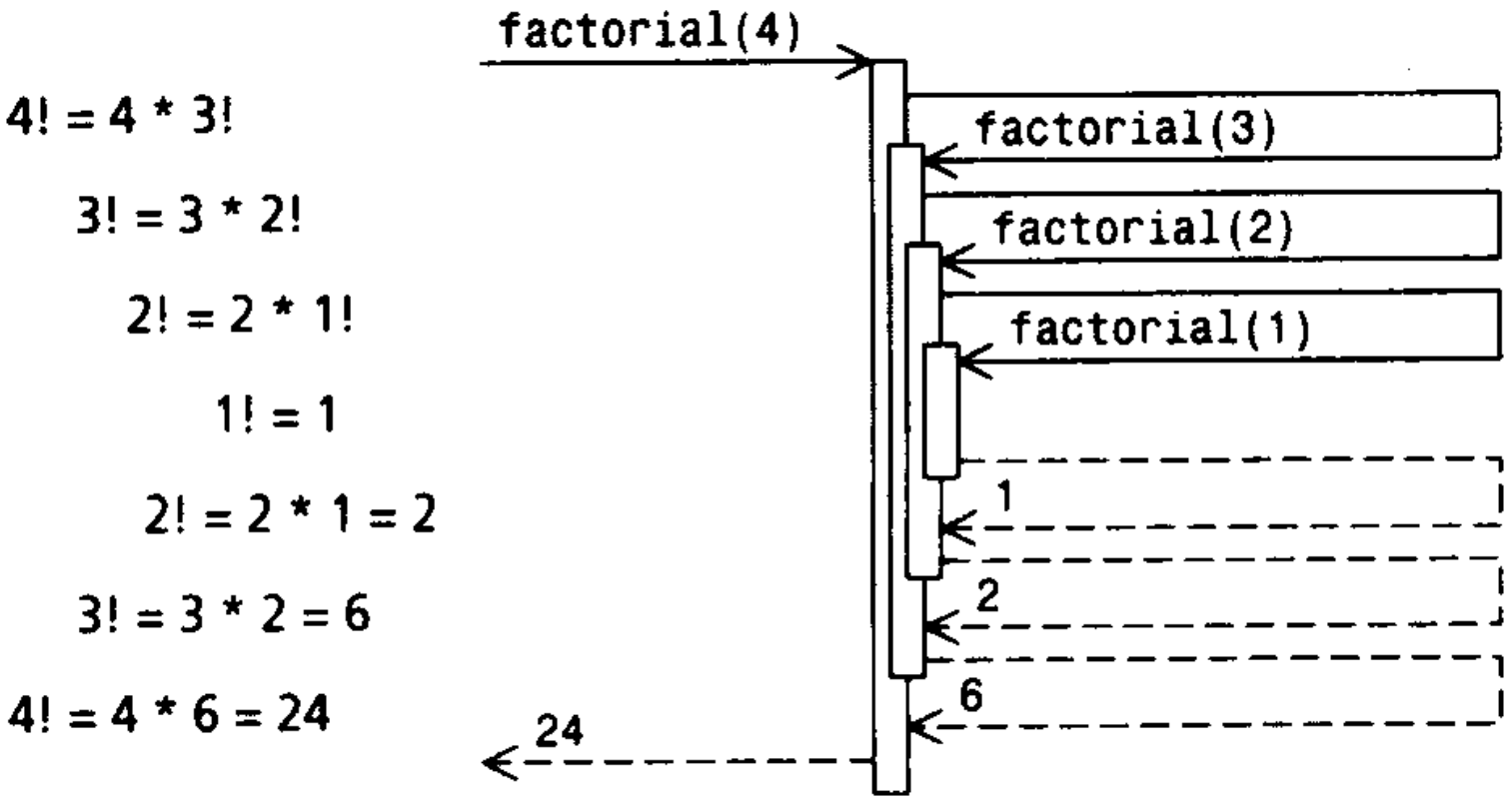
- Using a loop to compute the answer is called an iterative solution
- It is also possible to perform the same calculation using a recursive method in Java

```
1  // Recursive calculation of factorial numbers.
   import java.util.Scanner;
3  public class RecursiveFactorial {
     public static void main(String[] args) {
5      System.out.print("Type an integer to calculate its factorial: ");
       Scanner keyboard = new Scanner(System.in);
7      int n = keyboard.nextInt();

9      System.out.println(n + "! = " + factorial(n));
     }
11
     static int factorial(int n) {
13     if (n == 1)
         return 1;                          // 1! = 1        (base case)
15
       return n * factorial(n - 1);         // n! = n * (n-1)!
17   }
   }
```

- One definition of the factorial() method is not necessarily better than the other

- The recursive method directly implements the mathematical definition
- All methods that can be defined recursively can also be defined using iteration
- In many cases the recursive definition can be more elegant



Designing recursive algorithms

- A recursive agorithm solves a task by dividing it into smaller subtasks
- . . . . . using the same algorithm to solve each subtask until the task becomes trivial
- The parameters passed to a recursive method describe the extent of the task to be solved
- The arguments provided in the next recursive call describe a task that is smaller than the calling task

Designing recursive algorithms

The design of all recursive methods is based on:

Base cases:

- The parameters describe a simple task that is so trivial
- It is a task that can be solved directly (no further calls)
- A recursive method can contain more than 1 base case

- **GeneraL cases/Inductive Part**:
  - The parameters describe a task that can be divided into smaller subtasks
  - The subtasks are of the same kind as the overall task
  - Thy are then solved by calling the method recursively
  - The solution of the overall task is expressed by means of the solutions to the subtasks

Designing recursive algorithms

Problem

A path is 2 metres wide and n metres long. It is to be paved using paving stones of size $1m \times 2m$.
Write a recursive method that gives the number of ways can the paving be accomplished.

- **Base Cases:**
- **Inductive Part/Recursive Part/General Cases:**

Designing recursive algorithms


```
// Recursive calculation of the number of ways can the paving be
    accomplished.
import java.util.Scanner;
public class Paving {
  public static void main(String[] args) {
    System.out.print("Type the length of the path to pave: ");
    Scanner keyboard = new Scanner(System.in);
    int n = keyboard.nextInt();

    System.out.println("The paving can be accomplished in " +
        numberOfWays(n) + " ways.");
  }

  static int numberOfWays(int n) {
      int number;
      //System.out.printf("Calling now with n = %d %n", n);
      if (n == 1){
        number = 1;                          // p(1) = 1       (base case)
      } else {
          if (n == 2){
          number = 2;                        // p(2)= 2        (base case)
          } else {     // p(n) = p(n-1) + p(n-2)
              number = numberOfWays(n-1) + numberOfWays(n-2);
          }}
      return number;
  }
}
```

Program Output

```
Type the length of the path to pave: 35
The paving can be accomplished in 14930352 ways.
```

Infinite recursion

- For each recursive method call, the arguments get one step closer to the arguments of base case
- There should be a convergence towards a base case
- The recursion depth is the number of nested method calls that are necessary before a base case is reached
- If a recursive method does not converge towards a base case, the recursion will never stop
- Such a recursion is often called infinite recursion

Advantages and limitations of recursion

- **Advantages**
  - In some situations in programming, it is a must to use recursion (for simplicity of the program)
  - The recursion is very flexible in data structure like stacks, queues, linked list and quick sort
  - The length of the program can be reduced
  - The function mathematical definition is almost the program
- **Disadvantages**
  - It requires extra storage space (recursive calls and automatic variables are stored on the stack)
  - The recursion function is not efficient in execution speed and time