Method overloading

Example: method overloading

```java
public class SquaresApp {

    public static void main(String[] args) {

        int n = 6;

        for (int i=2; i<=n; i++){

            System.out.println(i + " " + MyMath.squre(i));
        }
```

```java
public class SquaresDoubleApp {

    public static void main(String[] args) {
      int n = 6;
      double x = 0.5;

      for (int i=2; i<=n; i++){
        double y = x + i;
        System.out.println(i + " " + MyMath.square(y));
      }
    }
}
```

public class MyMath
public static int random(int N) public static **int** square(**int** N) public static **double** square(**double** N)

```java
public class MyMath {

  public static int randomInt(int N) {
    double x = Math.random();
    int rndInt = (int) (x*N)+1;
    return rndInt;
  }
  public static int square(int N) {
    return N*N;
  }
  public static double square(double N) {
    return N*N;
  }
}
```

Method overloading

- The same method name with different arguments
  - Compiler understands the meanings from their different arguments

public class MyMath
public static int random(int N) public static int square(int N) public static double square(double N)

- We overload words in natural languages all the time
  - The 3 "open" attributes to different *actions* using different *methods* to produce different *results*

Open the door

Open the book

Open the computer file

```java
//              MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public static void main(String[] args)
   {
      System.out.printf("Square of integer 7 is %d%n", square(7));
      System.out.printf("Square of double 7.5 is %f%n", square(7.5));
   }

   // square method with int argument
   public static int square(int intValue)
   {
      System.out.printf("%nCalled square with int argument: %d%n",
         intValue);
      return intValue * intValue;
   }

```

Overloaded method declarations. (Part 1 of 2.)

```java
   // square method with double argument
   public static double square(double doubleValue)
   {
      System.out.printf("%nCalled square with double argument: %f%n",
         doubleValue);
      return doubleValue * doubleValue;
   }
} // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Overloaded method declarations. (Part 2 of 2.)

Method Overloading

- Method overloading
  - Methods of the same name declared in the same class
  - Must have different sets of parameters
- Compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.
- In the example
  - Literal integer values are treated as type int, so the method call in line 9 invokes the version of square that specifies an int parameter.
  - Literal floating-point values are treated as type double, so the method call in line 10 invokes the version of square that specifies a double parameter.

Method Overloading

Distinguishing Between Overloaded Methods

•The compiler distinguishes overloaded methods by their signatures—the methods' *name* and the *number*, *types* and *order* of its parameters.

•Return types of overloaded methods

- *Method calls cannot be distinguished by return type.*

•Overloaded methods can have different return types if the methods have different parameter lists.

•Overloaded methods need not have the same number of parameters.

Method-Call Stack and Stack Frames

- Stack data structure
  - Analogous to a pile of dishes
  - A dish is placed on the pile at the top (referred to as pushing the dish onto the stack).
  - A dish is removed from the pile from the top (referred to as popping the dish off the stack).
- Last-in, first-out (LIFO) data structures
  - The *last* item pushed onto the stack is the *first* item popped from the stack.

Method-Call Stack and Activation Records

- When a program *calls* a method, the called method must know how to *return* to its caller
  - The *return address* of the calling method is *pushed* onto the method-call stack.
- If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order.
- The method call stack also contains the memory for the *local variables* (including the method parameters) used in each invocation of a method during a program's execution.
  - Stored as a portion of the method call stack known as the stack frame (or activation record) of the method call.

Method-Call Stack and Activation Records

- When a method call is made, the stack frame for that method call is *pushed* onto the method call stack.
- When the method returns to its caller, the stack frame is *popped* off the stack and those local variables are no longer known to the program.
- If more method calls occur than can have their stack frames stored on the program-execution stack, an error known as a stack overflow occurs.

Recursion

Repetitive Processes

- Is it possible to implement a repetitive process without `while, do, for` loops?
- For example, using a solution where a function calls itself

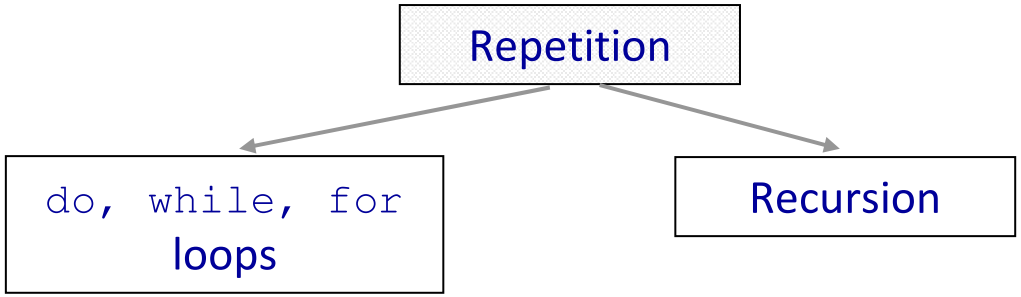
```
float myFunction( float x )
{   .    .    .
   temp = myFunction( y );
   .   .    .
   return result;
}
```

This may result in repetition, but...

Can this approach create an infinite loop as the function cannot reach return ?

Yes, unless implemented according to certain rules

Recursion



- <u>Recursion is not a trivial function call of itself</u>
- The idea is that a problem can be solved by reducing it to smaller versions of itself

Iteration

- Consider the calculation of `n!`, factorial `n`:

Given $n \geq 0$:

$$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1$$

$$0! = 1,\ 1! = 1$$

Example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Iteration

- Consider a factorial function, that uses a loop to calculate factorial iteratively

$$5! = 5 \times 4 \times 3 \times 2 \times 1 \times 1$$

1. 0! = 1
2. Loop

$1*2 = 2$	:	*first iteration*
$2*3 = 6$	:	*second iteration*
$6*4 = 24$	:	*third iteration*
$24*5 = 120$	:	*forth iteration*

Iteration

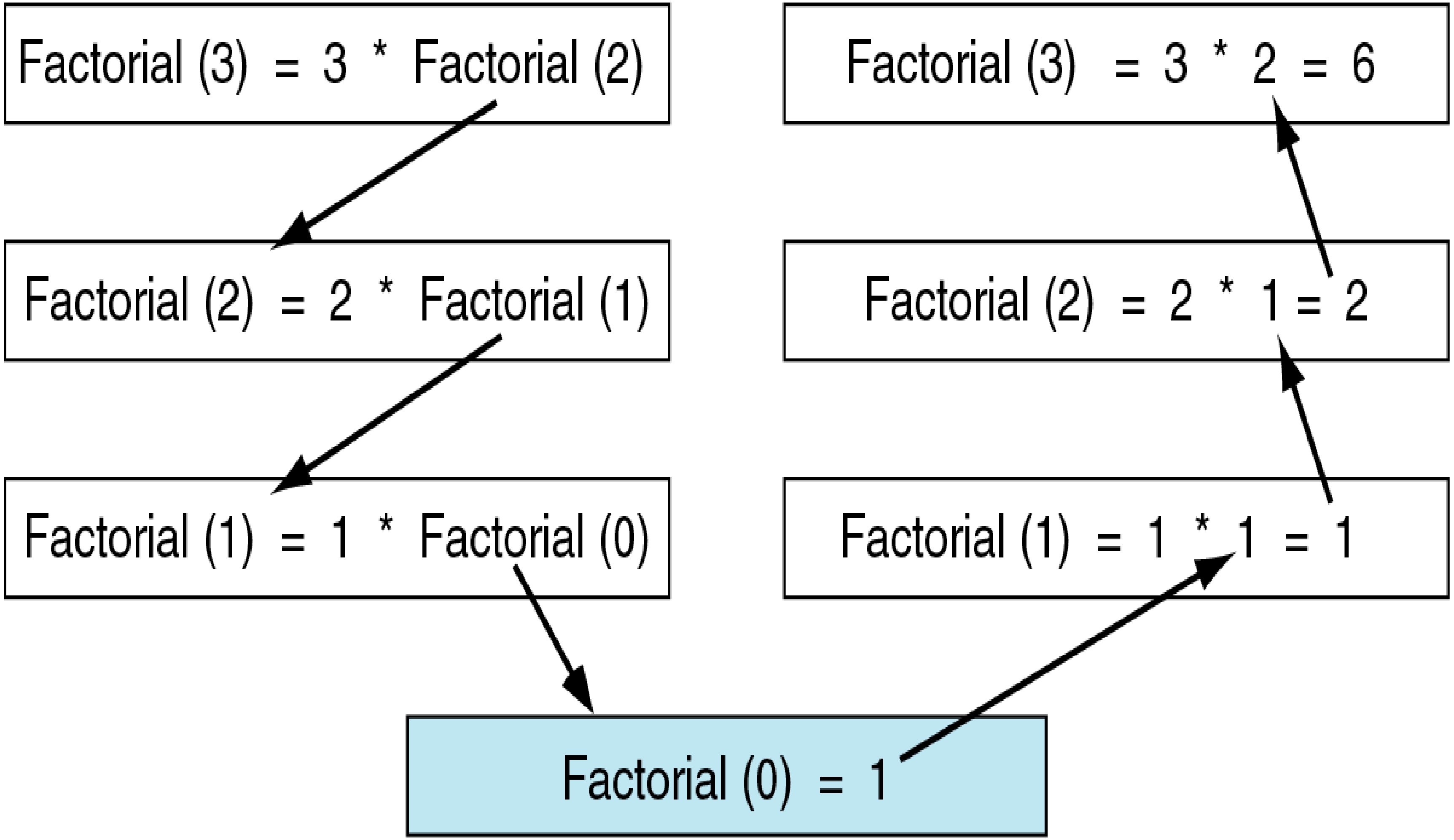
```
int factorial (int num)
{
  int i;
  int factN = 1;  /* 0! = 1  */

  for( i = 1; i <= num; i++)
     factN *= i;  /* iterations */
  return factN;
}
```

Recursion

- Recursive definition for factorial function:
  - factorial(n) = 1, if n = 0.
  - factorial(n) = n * factorial(n-1), if n>0.

```
factorial(5) = 5* factorial(4)
where factorial(4) = 4*factorial(3)
 where factorial(3) = 3*factorial(2)
  where factorial(2) = 2*factorial(1)
   where factorial(1) = 1*factorial(0)
    where factorial(0) = 1
```

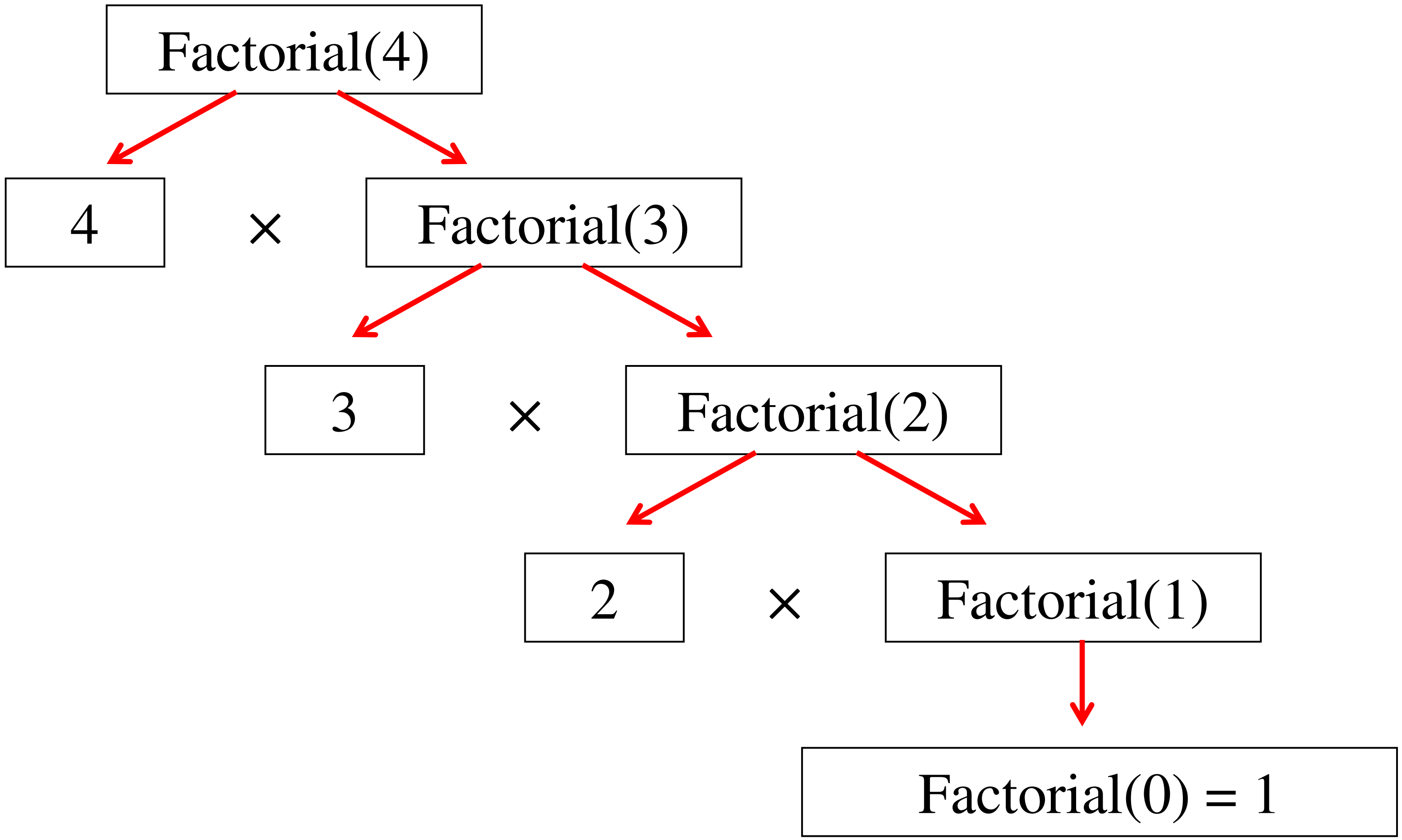
Recursion


Factorial (3) = 3 * Factorial (2)
Factorial (2) = 2 * Factorial (1)
Factorial (1) = 1 * Factorial (0)
Factorial (0) = 1
Factorial (1) = 1 * 1 = 1
Factorial (2) = 2 * 1 = 2
Factorial (3) = 3 * 2 = 6

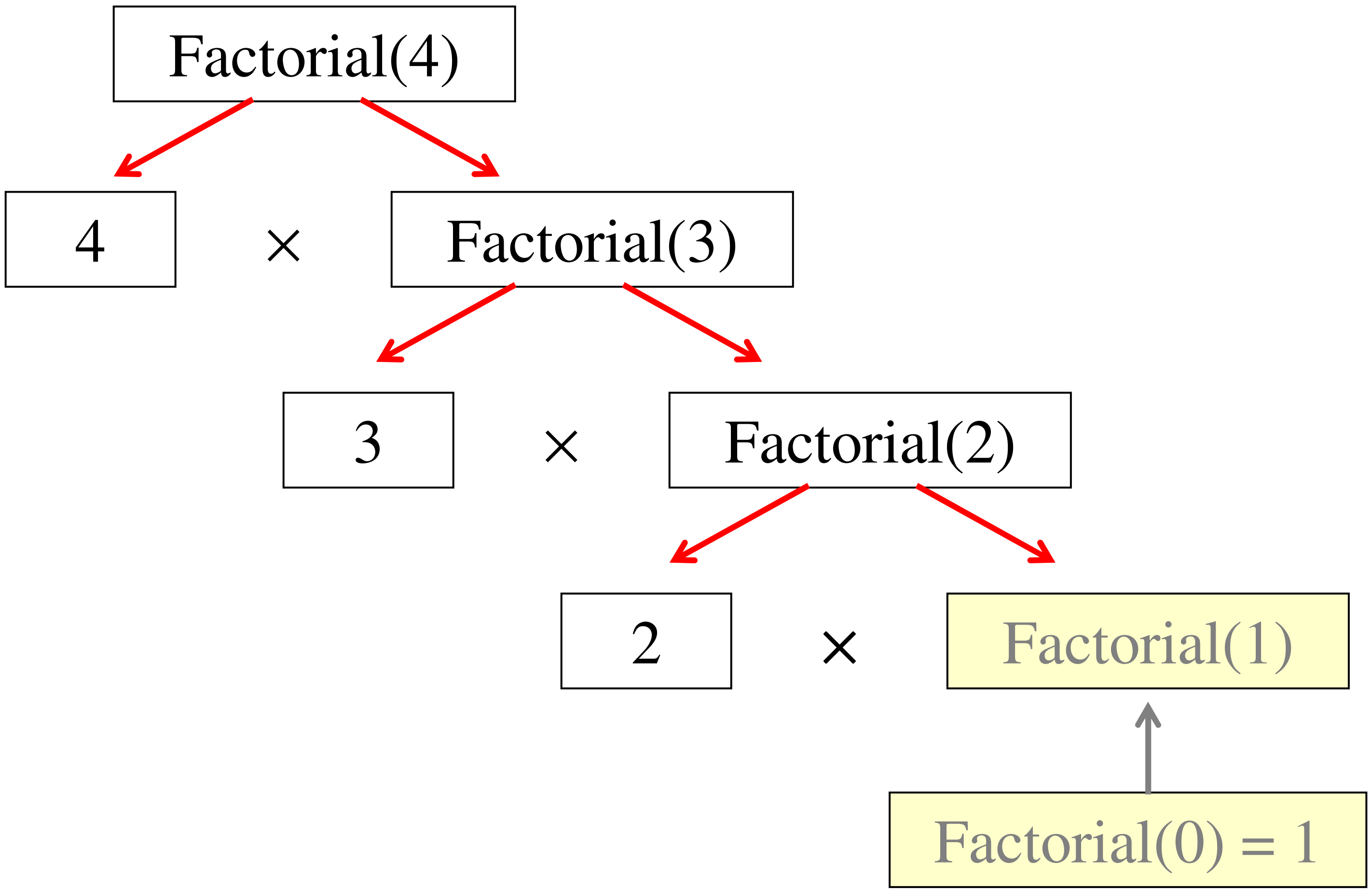
Recursion

```
int factorial( int num )
{
  if (num == 0)
      return 1;
  else
      return ( num * factorial(num-1) );
}
```

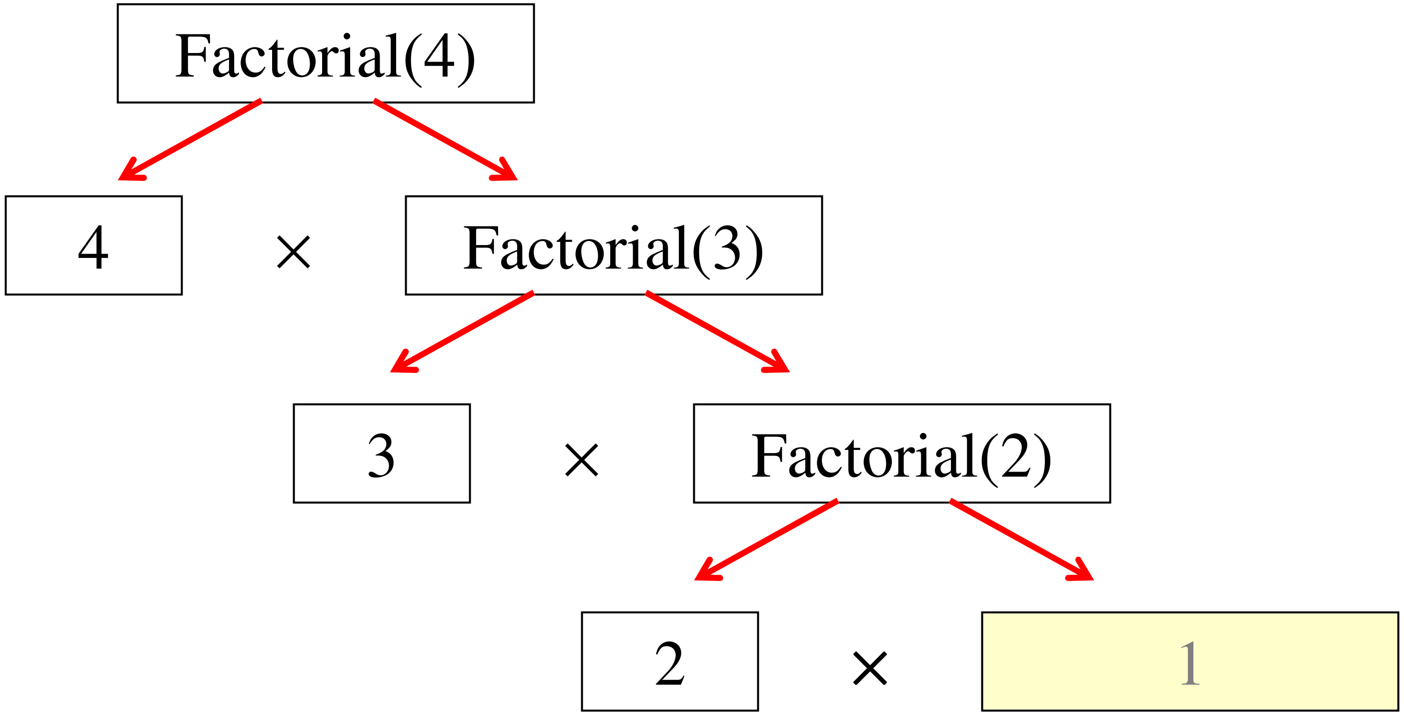
Recursion


Factorial(4)
4
×
Factorial(3)
3
×
Factorial(2)
2
×
Factorial(1)
Factorial(0) = 1

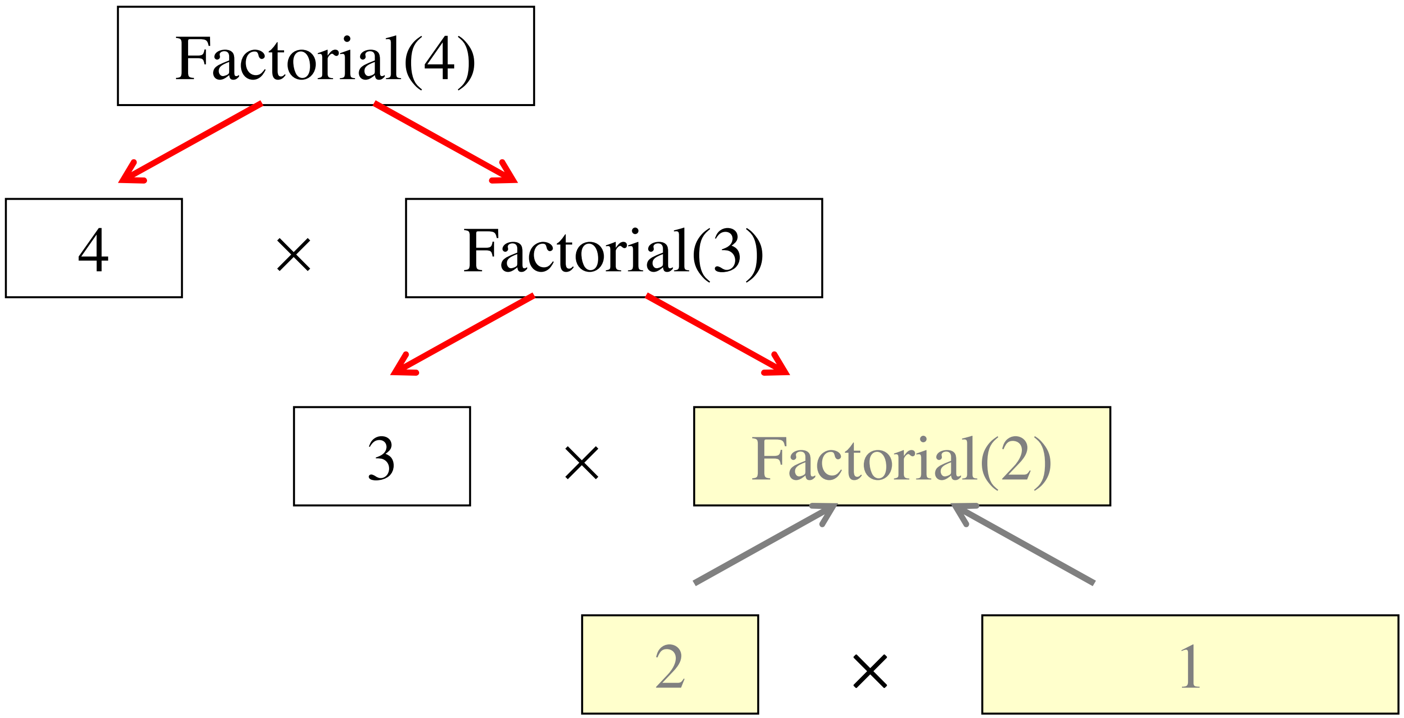
Recursion


Factorial(4)
4
×
Factorial(3)
3
×
Factorial(2)
2
×
Factorial(1)
Factorial(0) = 1

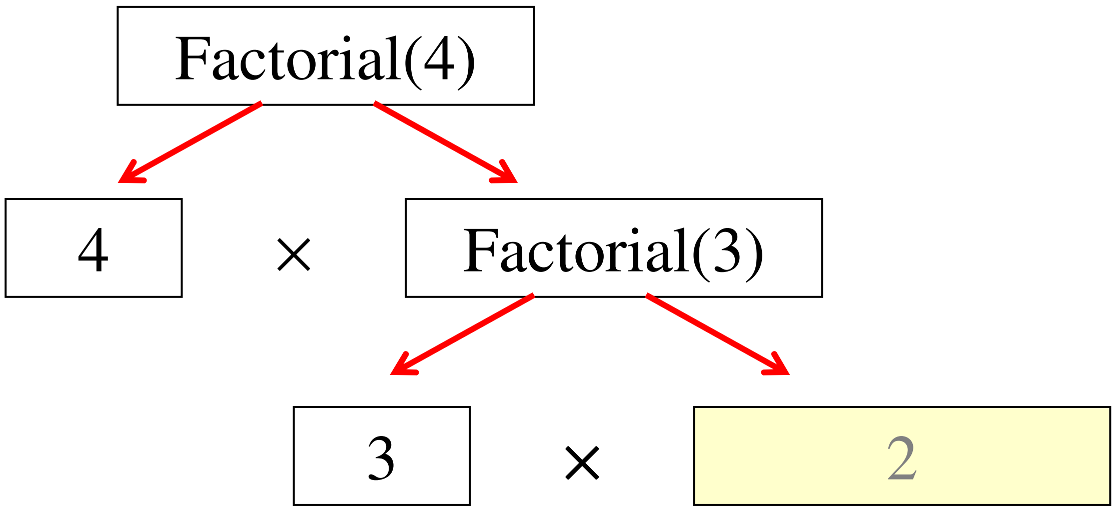
Recursion



Recursion

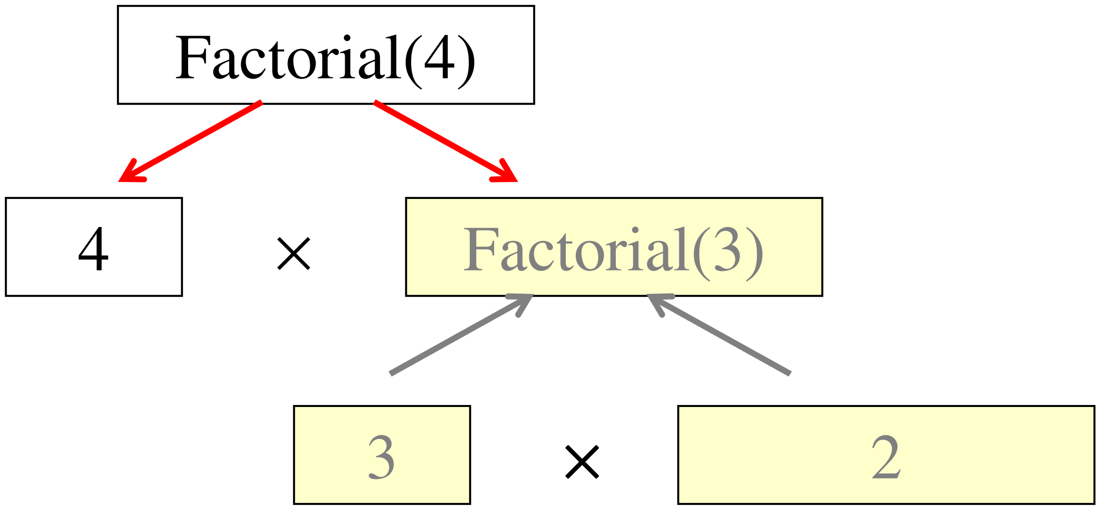


Recursion



UNIVERSITY OF WOLLONGONG

Recursion

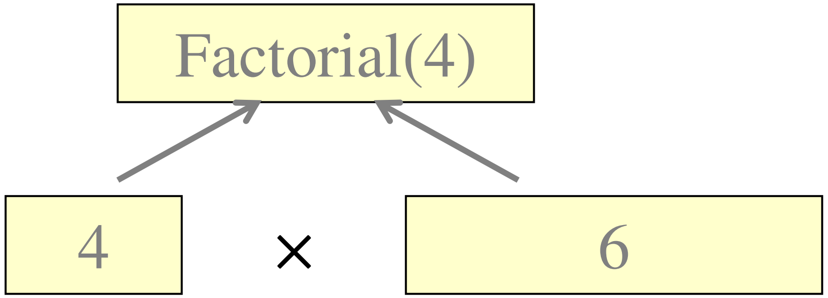


Recursion

Factorial(4)

4×6

Recursion



Recursion

24

Recursive Function Structure

- Every recursive function has two components:

1 The *base case* : the statement that “solves” the problem. The *base case* is also known as the *stopping condition*.

Example: `if(n==0) factN = 1;`

2 The *recursion step (recursive call)*: the statement that reduces the size of the problem

Example:

```
if(n != 0) factN = n * factorial(n-1);
```

Recursive function design

1. Determine the *base case*.
2. Then determine the *recursive step*.
3. Combine the base case and recursive step into a function.
4. In combining the base case and recursive step into a function, you must pay careful attention to the logic.

   - Each call <u>must reduce</u> the size of the problem and move it toward the base case.

   - The base case, when reached, must terminate without a function call. It must execute a *return*.

Recursion Examples

- Task: Create a function `power()` to compute x^n. Where n is a positive number

1. Create an iterative version of the function
2. Create a recursive version of the function
3. Compare two solutions

Iterative solution

$$x^n = \underbrace{x * x * x \ldots * x}_{\text{n times}}$$

```
/* Iterative solution */
double power(double x, int n)
{
    int i;
    double result = x;

    if (n == 0) return 1.0;

    for ( i=1; i < n; i++ )
        result *= x;

    return result;
}
```

Recursive solution

1. Base case

$x^0 = 1.0$

```
power(x, 0) = 1.0, if (n == 0)
```

2. Repetition step / General case

$x^n = x^* x^{n-1}$

```
  power(x, n) = x * power(x, n-1)
```

```
/* Recursive solution */
double power(double x, int n)
{
  if (n == 0)          /* base case */
      return 1.0;

  else                 /* repetition step */
      return ( x * power(x, n-1) );
}
```

Example: Calculate the sum

Write a program to calculate the sum from 1 to 10.

Loop solutions:

```
public void calculateSumFor() {
          int sum=0;
          for (int i=1;i<=10;i++){
            sum = sum + i;
          }
          System.out.println("The sum from 1 to 10 is "+sum);
}
```

```
public void calculateSumWhile() {
          int sum=0, i=1;
          while(i <= 10){
            sum = sum +i;
            i = i + 1;
          }
          System.out.println("The sum from 1 to 10 is "+sum);
}
```

```
public void calculateSumDoWhile() {
          int sum=0, i=1;
          do{
            sum = sum + i;
            i= i + 1;
          }while(i <= 10)
          System.out.println("The sum from 1 to 10 is "+sum);
}
```

Example: Calculate the sum

Write a program to calculate the sum from 1 to 10.

Recursion solution:

```
public static int calculateSumRecursion(int n) {
          if (n==1)
            return 1;
          else
            return (n + calculateSumRecursion(n-1));
}

public static void main(String args[]){
          System.out.println("The sum from 1 to 10 is " + calculateSumRecursion (10));
}
```

```
The sum from 1 to 10 is 55
------
n=10, calculateSumRecursion(10) Return (10+calculateSumRecurision(9));
n=9, calculateSumRecursion(9) Return (9+calculateSumRecurision(8));
n=8, calculateSumRecursion(8) Return (8+calculateSumRecurision(7));
n=7, calculateSumRecursion(7) Return (7+calculateSumRecurision(6));
n=6, calculateSumRecursion(6) Return (6+calculateSumRecurision(5));
n=5, calculateSumRecursion(5) Return (5+calculateSumRecurision(4));
n=4, calculateSumRecursion(4) Return (4+calculateSumRecurision(3));
n=3, calculateSumRecursion(3) Return (3+calculateSumRecurision(2));
n=2, calculateSumRecursion(2) Return (2+calculateSumRecurision(1));
n=1, calculateSumRecursion(1) Return 1;
```

Iterative vs. Recursive

- In general, recursive solutions are shorter and look more "elegant"
- Recursion can considerably simplify implementation of some commonly used algorithms
- If an iterative solution is more obvious and easier to understand than a recursive solution, use the iterative solution
- There is overhead associated with executing a recursive function both in terms of memory space and execution time.
- A recursive function executes more slowly than its iterative counterpart

Suggested reading

Java: How to Program (Early Objects), 10th Edition

- Chapter 6: Methods: A Deeper Look
  - 6.6
- Chapter 18: recursion
  - 18.1-18.3