Cape Peninsula University of Technology

Software Design 2
SDN260S

Recursion

H. Mataifa

Department of Electronic, Electrical and Computer Engineering

Outline

- Concept of recursion
- Writing and using recursive methods
- Determining the base case and recursion step in a recursive algorithm
- Differences between recursion and iteration

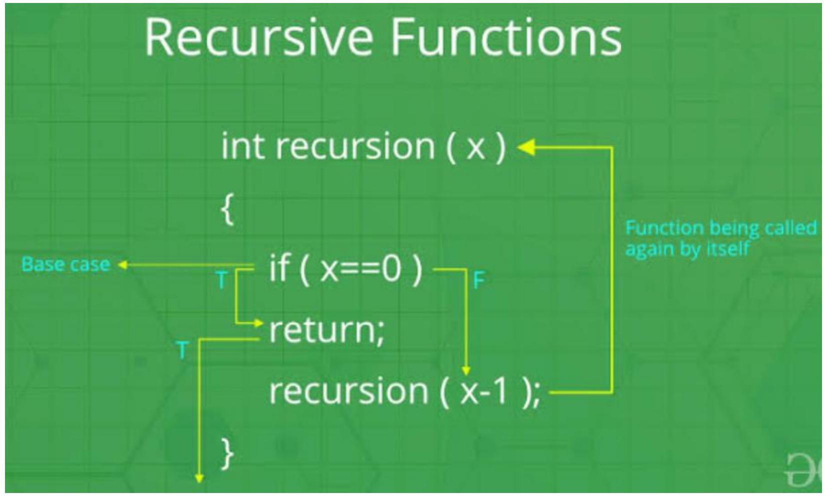
Basic Concept of Recursion

- Recursive method: *a method that calls itself*, either directly or indirectly (i.e. through another method)

- Uses the concept of breaking up a problem into simpler (smaller) problems, each of which is a replica of itself

- Can only solve the simplest subproblem, referred to as the base case; thus, the recursive method calls must eventually arrive at the base case
  - To eventually arrive at the base case, every subsequent recursive method call (to itself) should be a reduced version

- Once the base case is reached, a sequence of returns follows, starting from the base case, going backwards until the original recursive method call



Recursive method

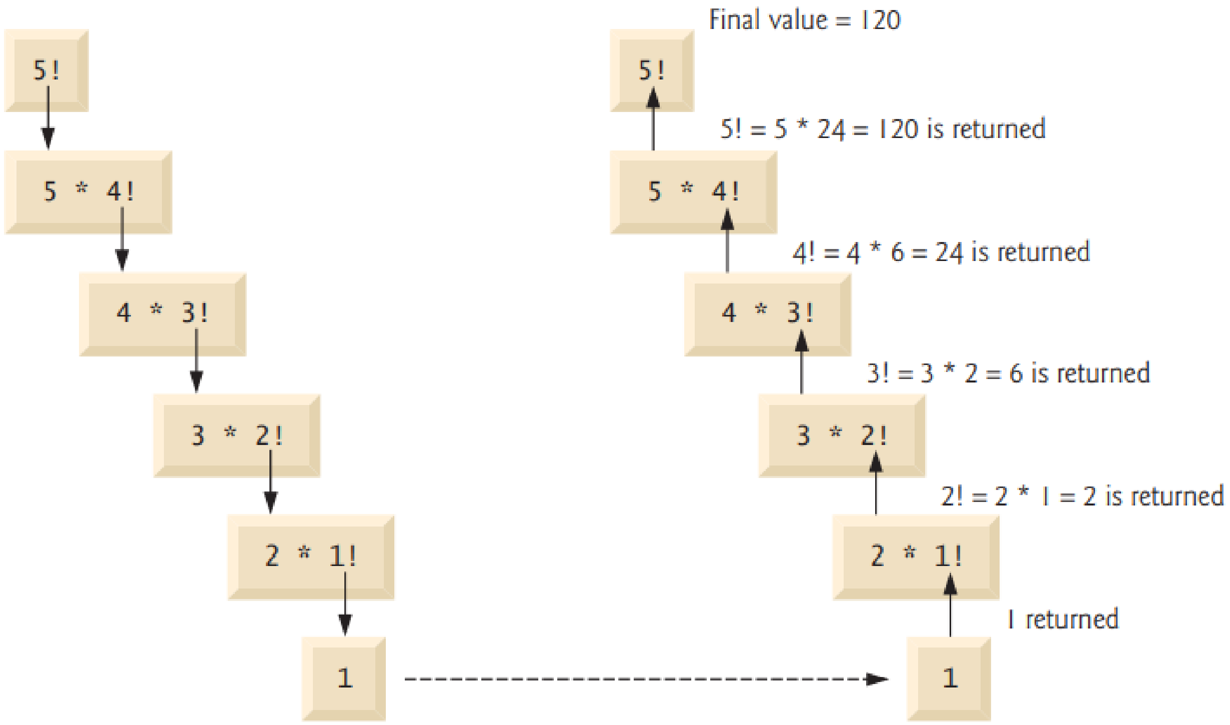
Recursive Method Example 1: Factorial

- Factorial of a positive integer, n (n!): a product n*(n-1)*…*1 (with 0!=1!=1)
- We notice that for any given n, n!=n*(n-1)!
- Program in Fig. 18.3 defines a recursive method for the factorial of **n**

```java
// Fig. 18.3: FactorialCalculator.java
// Recursive factorial method.

public class FactorialCalculator
{
   // recursive method factorial (assumes its parameter is >= 0)
   public long factorial( long number )
   {
      if ( number <= 1 ) // test for base case
         return 1; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number * factorial( number - 1 );
   } // end method factorial

   // output factorials for values 0-21
   public static void main( String[] args )
   {
      // calculate the factorials of 0 through 21
      for ( int counter = 0; counter <= 21; counter++ )
         System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
   } // end main
} // end class FactorialCalculator
```

- Program in Fig. 18.4 defines the same function, but uses BigInteger, which enables working with much larger numbers

Recursive Method Example 1: Factorial



(a) Sequence of recursive calls

(b) Values returned from each recursive call

Computing 5! recursively

```
// recursive method factorial (assumes its parameter is >= 0)
public long factorial( long number )
{
   if ( number <= 1 ) // test for base case
      return 1; // base cases: 0! = 1 and 1! = 1
   else // recursion step
      return number * factorial( number - 1 );
} // end method factorial
```

Recursive Method Example 2: Fibonacci Series

- Fibonacci series: a special mathematical sequence of numbers, starting with 0, 1, with the property that every subsequent number (following 1) is the *sum of the previous two*

- So the Fibonacci numbers (except for the first two) are defined in terms of other (previous) Fibonacci numbers; thus we are able to apply recursion to the generation of the Fibonacci series

- The first 10 Fibonacci numbers are: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34,…}

- Interesting facts about the Fibonacci series:
  - Occurs in many instances in nature (e.g. describes a form of a spiral)
  - Ratio of successive Fibonacci numbers converges to constant value of 1.618… (referred to as the golden ratio/mean)

- Recursive definition of Fibonacci series:
  - *fibonacci(0)=0*
  - *fibonacci(1)=1*
  - *fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)*

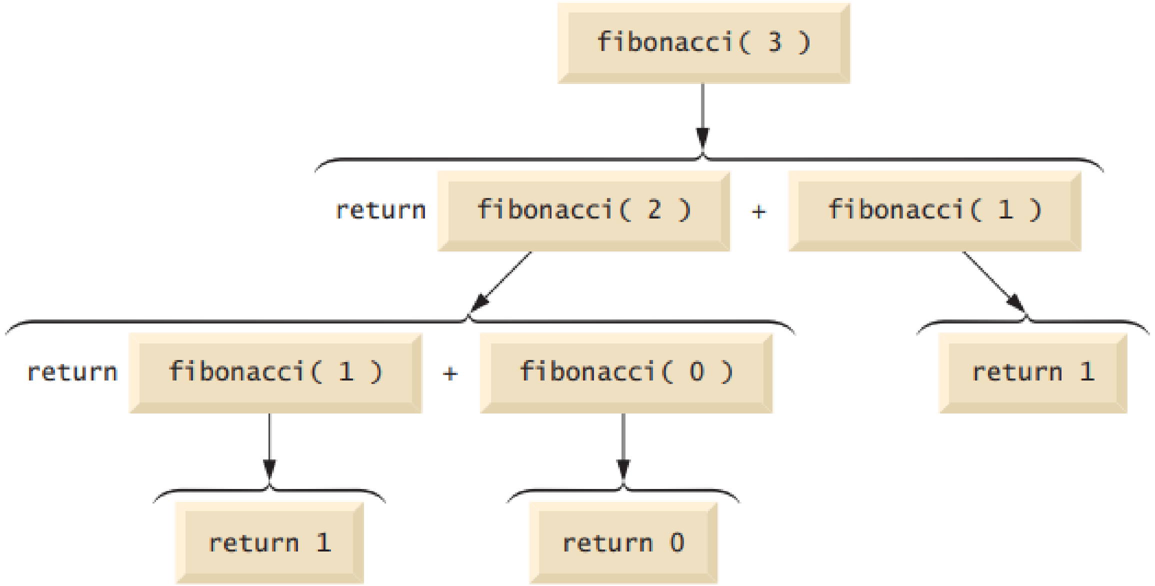
- Recursive definition of Fibonacci series has two base cases: {*fibonacci(0)=0, and fibonacci(1)=1* }

- BigInteger used, because Fibonacci numbers tend to become large quickly

Recursive Method Example 2: Fibonacci Series

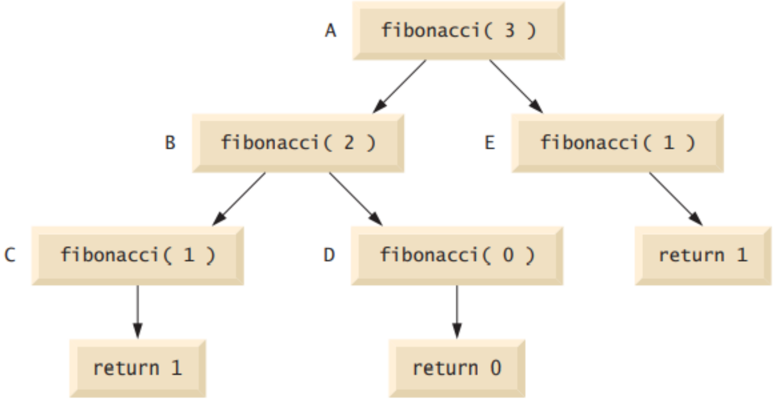
```java
// Fig. 18.5: FibonacciCalculator.java
// Recursive fibonacci method.
import java.math.BigInteger;

public class FibonacciCalculator
{
   private static BigInteger TWO = BigInteger.valueOf( 2 );

   // recursive declaration of method fibonacci
   public static BigInteger fibonacci( BigInteger number )
   {
      if ( number.equals( BigInteger.ZERO ) ||
           number.equals( BigInteger.ONE ) ) // base cases
         return number;
      else // recursion step
         return fibonacci( number.subtract( BigInteger.ONE ) ).add(
            fibonacci( number.subtract( TWO ) ) );
   } // end method fibonacci

   // displays the fibonacci values from 0-40
   public static void main( String[] args )
   {
      for ( int counter = 0; counter <= 40; counter++ )
         System.out.printf( "Fibonacci of %d is: %d\n", counter,
            fibonacci( BigInteger.valueOf( counter ) ) );
   } // end main
} // end class FibonacciCalculator
```

Recursive Method Example 2: Fibonacci Series

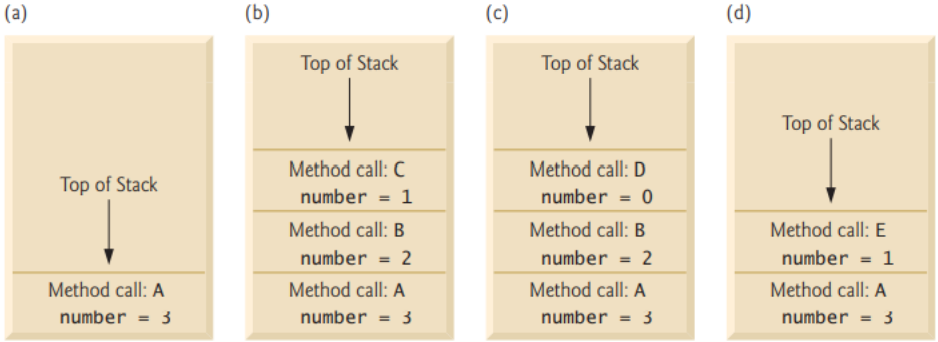


Recursive calls to fibonacci(3)

Recursion and Method-Call Stack



Method calls of fibonacci(3)



Method calls on program-execution stack

Method fibonacci(n)

```
// recursive declaration of method fibonacci
public static BigInteger fibonacci( BigInteger number )
{
   if ( number.equals( BigInteger.ZERO ) ||
        number.equals( BigInteger.ONE ) ) // base cases
      return number;
   else // recursion step
      return fibonacci( number.subtract( BigInteger.ONE ) ).add(
         fibonacci( number.subtract( TWO ) ) );
} // end method fibonacci
```

Recursion vs. Iteration

Both iteration and recursion make use of (i) a *control statement*, (ii) *repetition*, and (iii) a *termination test*:

- **Control statement**:
  - Iteration: uses repetition statement (for, while, do…while)
  - Recursion: uses selection statement (if, if…else, switch)
- **Repetition**:
  - Iteration: uses repetition statement explicitly
  - Recursion: achieves repetition through repeated method calls
- **Termination test**:
  - Iteration: terminates when loop-continuation condition fails
  - Recursion: terminates when a base case is reached
- Risk of infinite occurrence:
  - Iteration: *infinite loop* occurs when loop-continuation condition never fails
  - Recursion: *infinite recursion* occurs when base case is never reached

Recursion vs. Iteration

- Recursion can be very expensive, both in terms of processor time and memory space, because each recursive call requires new memory allocation
- Iteration occurs within a (single) method, thus requires no extra memory allocation
- Generally, any problem that can be solved recursively can also be solved iteratively (i.e. non-recursively)

```java
// Fig. 18.9: FactorialCalculator.java
// Iterative factorial method.

public class FactorialCalculator
{
   // recursive declaration of method factorial
   public long factorial( long number )
   {
      long result = 1;

      // iterative declaration of method factorial
      for ( long i = number; i >= 1; i-- )
         result *= i;

      return result;
   } // end method factorial

   // output factorials for values 0-10
   public static void main( String[] args )
   {
      // calculate the factorials of 0 through 10
      for ( int counter = 0; counter <= 10; counter++ )
         System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
   } // end main
} // end class FactorialCalculator
```

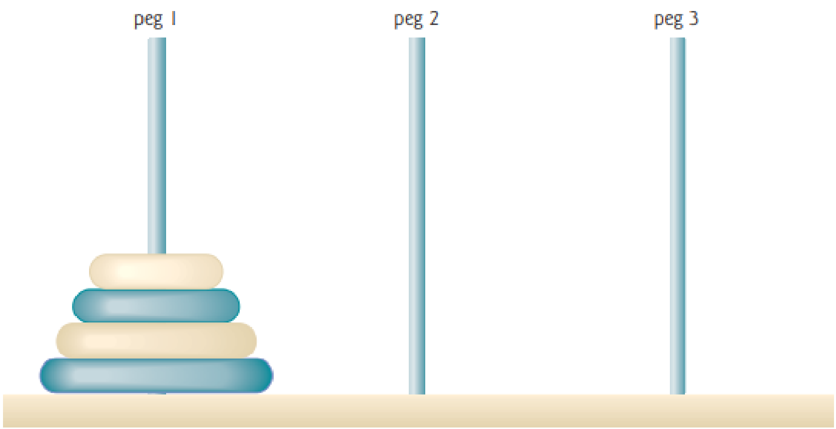

Iterative factorial(n) method

Recursion vs. Iteration

- **Use recursion when**: the recursive approach more naturally mirrors the problem, resulting in a problem that is easier to understand and to debug
  - A recursive solution can often be implemented with fewer lines of code
- **Avoid recursion when**: you care about execution time and memory requirement (i.e. high-performance requirement)
- **Note**: making a recursive call to a non-recursive method (i.e. having a non-recursive method call itself) can lead to infinite recursion (base case is never reached)

Recursive Method Example 3: Towers of Hanoi

- **Task**: move a stack of disks from one peg to another, with restrictions that:
  - (i) *exactly one disk is moved at a time*, and (ii) *a larger disk is never placed above a smaller one*
  - Three pegs provided, one being used for temporarily holding the disks
- Recursive solution to the problem: moving n disks viewed in terms of moving n-1 disks:
  - Move (top) n-1 disks from peg 1 to peg 2, using peg 3 as temporary holding area
  - Move the last (bottom) disk from peg 1 to peg 3
  - Move n-1 disks from peg 2 to peg 3, using peg 1 as temporary holding area
  - Process ends when last task involves moving n=1 disk (base case)



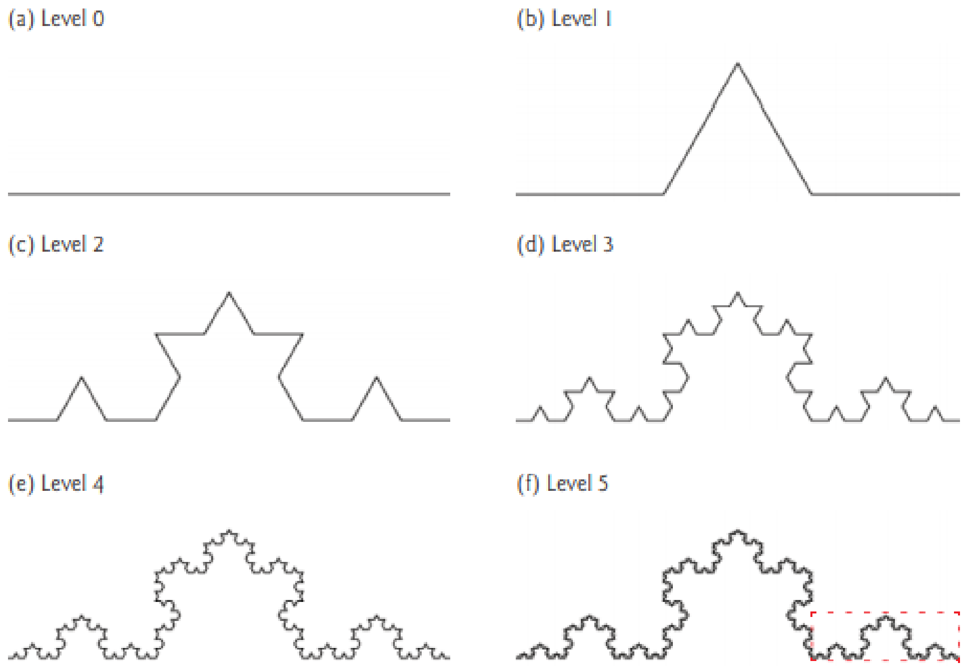
Four-disk Towers-of-Hanoi

Recursive Method Example 3: Towers of Hanoi

```java
// Fig. 18.11: TowersOfHanoi.java
// Towers of Hanoi solution with a recursive method.
public class TowersOfHanoi
{
   // recursively move disks between towers
   public static void solveTowers( int disks, int sourcePeg,
      int destinationPeg, int tempPeg )
   {
      // base case -- only one disk to move
      if ( disks == 1 )
      {
         System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg );
         return;
      } // end if

      // recursion step -- move (disk - 1) disks from sourcePeg
      // to tempPeg using destinationPeg
      solveTowers( disks - 1, sourcePeg, tempPeg, destinationPeg );

      // move last disk from sourcePeg to destinationPeg
      System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg );

      // move ( disks - 1 ) disks from tempPeg to destinationPeg
      solveTowers( disks - 1, tempPeg, destinationPeg, sourcePeg );
   } // end method solveTowers

   public static void main( String[] args )
   {
      int startPeg = 1; // value 1 used to indicate startPeg in output
      int endPeg = 3; // value 3 used to indicate endPeg in output
      int tempPeg = 2; // value 2 used to indicate tempPeg in output
      int totalDisks = 3; // number of disks

      // initial nonrecursive call: move all disks.
      towersOfHanoi.solveTowers( totalDisks, startPeg, endPeg, tempPeg );
   } // end main
} // end class TowersOfHanoi
```

Recursive Method Example 4: Fractals (Section 18.8)

- **Fractal**: a geometric figure that can be generated from a pattern repeated recursively, applying the pattern to each segment of the original figure
- Self-similar property of fractals: when subdivided into parts, each resembles a reduced-size copy of the whole (strictly self-similar if magnified subdivision is exact copy of original)
- Koch Curve fractal: bends middle third of a line segment so as to form an equilateral triangle; pattern repeated on each resulting line segment



Koch curve fractal

Assignment (recursive integer multiplication)
(Due Date: Tuesday, August 20th 2024)

- Design a Java application that will use a recursive method (recursiveMultiply(int,int)) to implement the multiplication arithmetic operation for any two integers; the following requirements apply:
  - Only the addition and subtraction arithmetic operators may be used in the definition of the recursiveMultiply method
  - The program will request the user to input the two integers to be multiplied, and will then compute the product and display the result on the console

- Following is a sample of calls to the method and the expected results:
  - recursiveMultiply(5,3)=15
  - recursiveMultiply(-5,3)=-15
  - recursiveMultiply(5,-3)=-15
  - recursiveMultiply(-5,-3)=15
  - recursiveMultiply(5,0)=0
  - recursiveMultiply(0,3)=0
  - recursiveMultiply(0,0)=0