Recursion
• Earlier programs structured as methods that call one another in a disciplined, hierarchical manner

• Recursive methods
  – Call themselves
  – Useful for some problems to define a method to call itself
  – Can be called directly or indirectly through another method
Recursion Concepts

- **Recursive problem-solving elements**
  - **Base case**
    - Recursive method capable of solving only simplest case—the base case
    - If method is called with base case, method returns result
  - If method is called with more complex problem, problem divided into two pieces—a piece the method knows how to do and a piece the method does not know how to do (called recursive call or recursion step)
  - **Recursive call/recursion step**
    - Must resemble original problem but be slightly simpler or smaller version
    - Method calls fresh copy of itself to work on smaller problem
    - Normally includes `return` statement

- **Indirect recursion**
  - Recursive method calls another method that eventually makes call back to recursive method
Example Using Recursion: Factorials

- Factorial of $n$, or $n!$ is the product
  
  $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$

  With $1!$ equal to 1 and $0!$ defined to be 1.

- Can be solved recursively or iteratively (nonrecursively)

- Recursive solution uses following relationship:
  
  $n! = n \cdot (n - 1)!$
Recursive evaluation of 5!

(a) Sequence of recursive calls.

(b) Values returned from each recursive call.
public class FactorialCalculator {

    // recursive method factorial
    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 0-10
    public void displayFactorials() {
        // calculate the factorials of 0 through 10
        for (int counter = 0; counter <= 10; counter++)
            System.out.printf("%d! = %d\n", counter, factorial(counter));
    } // end method displayFactorials

} // end class FactorialCalculator
// FactorialTest.java
// Testing the recursive factorial method.

public class FactorialTest
{
    // calculate factorials of 0-10
    public static void main( String args[] )
    {
        FactorialCalculator factorialCalculator = new FactorialCalculator();
        factorialCalculator.displayFactorials();
    }
} // end class FactorialTest

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
Example Using Recursion: Fibonacci Series

- Fibonacci series begins with 0 and 1 and has property that each subsequent Fibonacci number is the sum of previous two Fibonacci numbers.
- Series occurs in nature, ratio of successive Fibonacci numbers converges on golden ratio or golden mean
- Fibonacci series defined recursively as:
  
  \[
  \begin{align*}
  \text{fibonacci}(0) &= 0 \\
  \text{fibonacci}(1) &= 1 \\
  \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)
  \end{align*}
  \]
public class FibonacciCalculator {

    // recursive declaration of method fibonacci
    public long fibonacci( long number ) {
        if ( ( number == 0 ) || ( number == 1 ) ) // base cases
            return number;
        else // recursion step
            return fibonacci( number - 1 ) + fibonacci( number - 2 );
    } // end method fibonacci

    public void displayFibonacci() {
        for ( int counter = 0; counter <= 10; counter++ )
            System.out.printf( "Fibonacci of %d is: %d\n", counter, fibonacci( counter ) );
    } // end method displayFibonacci

} // end class FibonacciCalculator
// FibonacciTest.java
// Testing the recursive fibonacci method.

public class FibonacciTest {

    public static void main( String args[] ) {
        FibonacciCalculator fibonacciCalculator = new FibonacciCalculator();
        fibonacciCalculator.displayFibonacci();
    }

} // end class FibonacciTest
Set of recursive calls for \texttt{fibonacci(3)}.
Recursion and the Method Call Stack

- Method call stack used to keep track of method calls and local variables within a method call
- Just as with nonrecursive programming, recursive method calls are placed at the top of the method call stack
- As recursive method calls return, their activation records are popped off the stack and the previous recursive calls continue executing
- Current method executing is always method whose activation record is at top of stack
Method calls made within the call `fibonacci(3)`.
Method calls on the program execution stack.