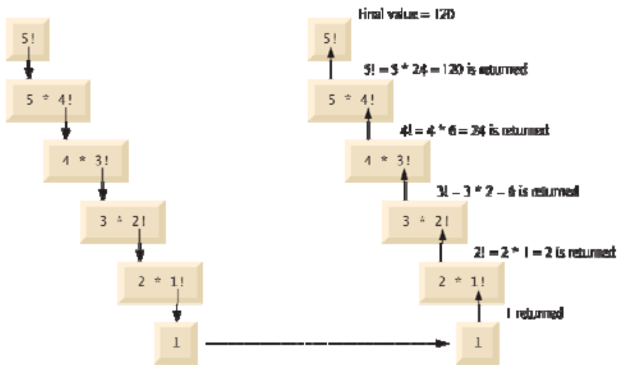
RECURSION

- For some problems, it's useful to have a method call itself.
  - Known as a recursive method.
  - Can call itself either directly or indirectly through another method.
- When a recursive method is called to solve a problem, it actually is capable of solving only the simplest case(s), or base case(s).
  - If the method is called with a base case, it returns a result.
- If the method is called with a more complex problem, it typically divides the problem into two conceptual pieces
  - a piece that the method knows how to do and
  - a piece that it does not know how to do.

- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version of it.
- Because this new problem looks like the original problem, the method calls a fresh copy of itself to work on the smaller problem
  - this is a recursive call (also called recursion step)
- The recursion step normally includes a `return` statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

- For recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must 'converge' on a base case.
  - When the method recognizes the base case, it returns a result to the previous copy of the method.
  - A sequence of returns continues until the original method call returns the final result to the caller.
- A recursive method may call another method, which may in turn make a call back to the recursive method.
  - This is known as an indirect recursive call or indirect recursion.

- Factorial of a positive integer *n*, written *n*! which 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.
- The factorial of integer `number` (where `number` ≥ 0) can be calculated iteratively (nonrecursively) using a `for` statement as follows:
  - `factorial = 1;`
  - ```
    for ( int counter = number; counter >= 1; counter-- )
       factorial *= counter;
    ```
- Recursive declaration of the factorial method is arrived at by observing the following relationship:
  - $n! = n \cdot (n - 1)!$
- Figure 18.3 uses recursion to calculate and print the factorials of the integers from 0 to 21.



(a) Sequence of recursive calls.   (b) Values returned from each recursive call.

Fig. 18.2 | Recursive evaluation of 5!.
    ```

```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
```

Recursive method call solves simpler problem

Nonrecursive method call

Fig. 18.3 | Factorial calculations with a recursive method. (Part 1 of 2.)

- The program uses type `long` so it can calculate factorials greater than 12!.
- The `factorial` method produces large values so quickly that we exceed the largest `long` value when we attempt to calculate 21!.
- Package `java.math` provides classes `BigInteger` and `BigDecimal` explicitly for arbitrary precision calculations that cannot be performed with primitive types.

```java
// Fig. 18.4: FactorialCalculator.java
// Recursive factorial method.
import java.math.BigInteger;

public class FactorialCalculator
{
   // recursive method factorial (assumes its parameter is >= 0)
   public static BigInteger factorial( BigInteger number )
   {
      if ( number.compareTo( BigInteger.ONE ) <= 0 ) // test base case
         return BigInteger.ONE; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number.multiply(
            factorial( number.subtract( BigInteger.ONE ) ) );
   } // end method factorial

   // output factorials for values 0-50
   public static void main( String[] args )
   {
      // calculate the factorials of 0 through 50
      for ( int counter = 0; counter <= 50; counter++ )
         System.out.printf( "%d! = %d\n", counter,
            factorial( BigInteger.valueOf( counter ) ) );
   } // end main
} // end class FactorialCalculator
```

Supports arbitrarily large integers

Fig. 18.4 | Factorial calculations with a recursive method. (Part 1 of 2.)

- `BigInteger` method `compareTo` compares the `BigInteger` that calls the method to the method's `BigInteger` argument.
  - Returns `-1` if the `BigIteger` that calls the method is less than the argument, `0` if they are equal or `1` if the `BigInteger` that calls the method is greater than the argument.
- `BigInteger` constant `ONE` represents the integer value 1.
- `BigInteger` methods `multiply` and `subtract` implement multiplication and subtraction. Similar methods are provided for other arithmetic operations.