Recursion

Recursive Definitions

- Recursion
  - Process of solving a problem by reducing it to smaller versions of itself
- Recursive definition
  - Definition in which a problem is expressed in terms of a smaller version of itself
  - Has one or more base cases

Recursive Definitions (Cont'd)

0! = 1 (By Definition!)
n! = n x (n – 1) ! If n > 0
3! = 3 x 2!
2! = 2 x 1!
1! = 1 x 0!

0! = 1 (Base Case!)

1! = 1 x 0! = 1 x 1 = 1
2! = 2 x 1! = 2 x 1 = 2
3! = 3 x 2! = 3 x 2 = 6

Recursive Definitions (Cont'd)

- Recursive algorithm
  - Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself
  - Has one or more base cases
  - Implemented using recursive methods
- Recursive method
  - Method that calls itself
- Base case
  - Case in recursive definition in which the solution is obtained directly
  - Stops the recursion

Recursive Definitions (Cont'd)

- General solution
  - Breaks problem into smaller versions of itself
- General case
  - Case in recursive definition in which a smaller version of itself is called
  - Must eventually be reduced to a base case

Tracing a Recursive Method

- Recursive method
  - Logically, you can think of a recursive method having unlimited copies of itself
  - Every recursive call has its own
    - Code
    - Set of parameters
    - Set of local variables

Designing Recursive Methods

- Understand problem requirements
- Determine limiting conditions
- Identify base cases

Designing Recursive Methods

- Provide direct solution to each base case
- Identify general case(s)
- Provide solutions to general cases in terms of smaller versions of general cases

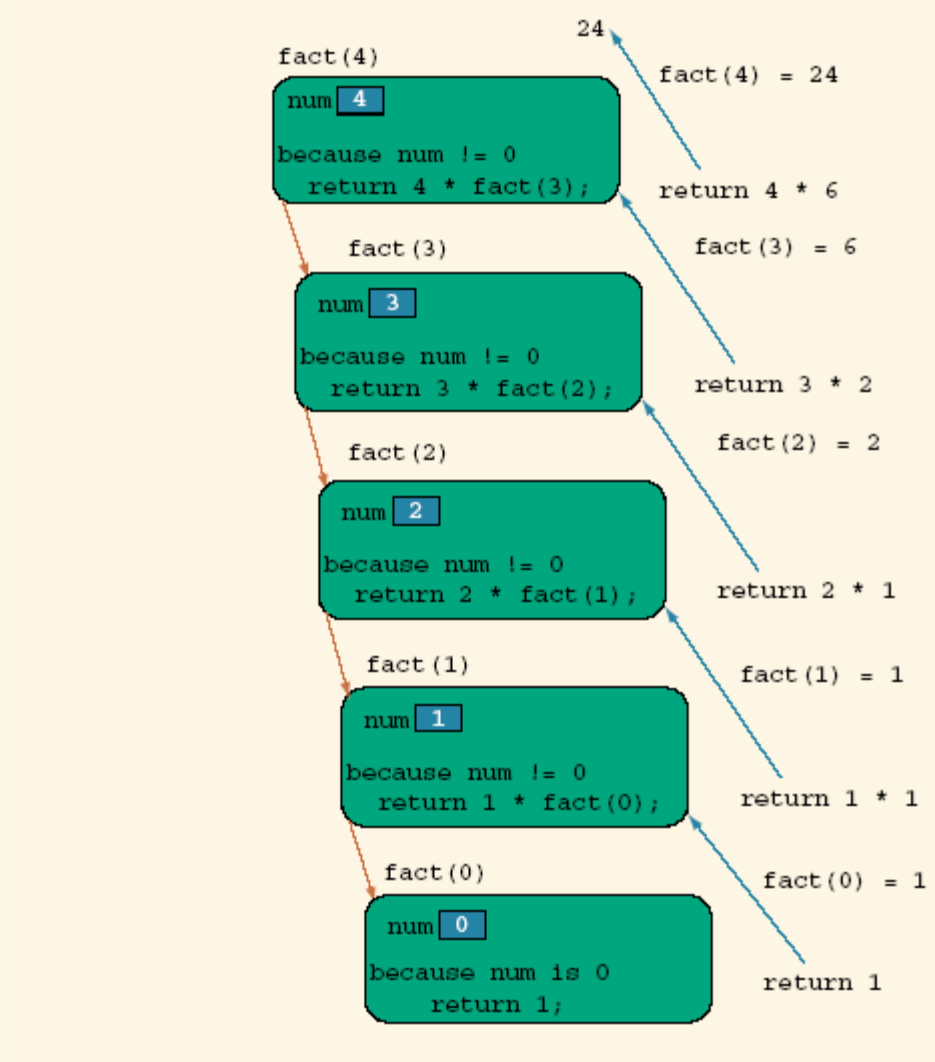
Recursive Factorial Method

```
public static int fact(int num) {
    if (num = = 0)
        return 1;
    else
        return num * fact(num – 1);
}
```

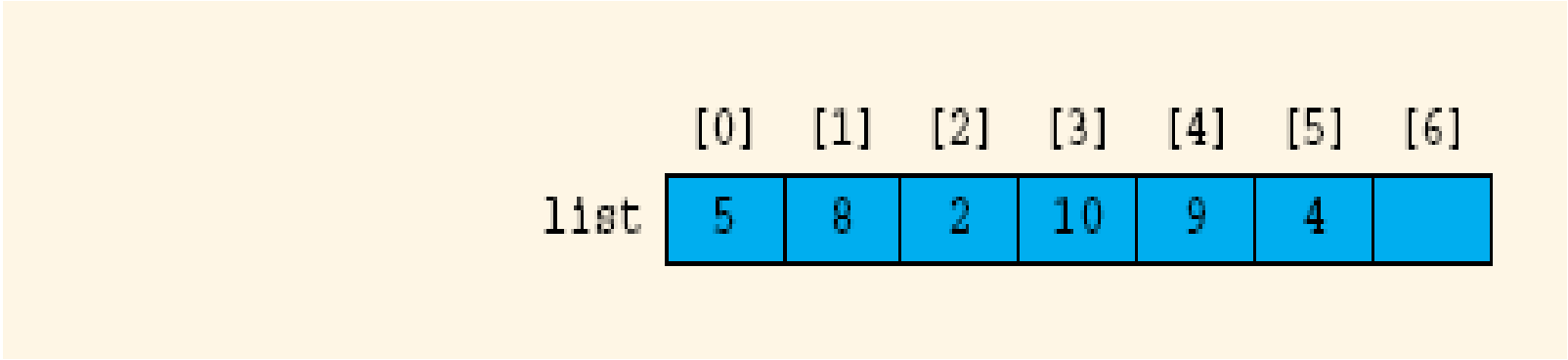
0! = 1 (Base Case!)

1! = 1 x 0! = 1 x 1 = 1
2! = 2 x 1! = 2 x 1 = 2
3! = 3 x 2! = 3 x 2 = 6

Recursive Factorial Method



Largest Value in Array



Largest Value in Array

`if` the size of the list is `1` - the largest element in the list is the only element in the array

`else` - to find the largest element in list [a] .. list[b]

a. find the largest element in list[a+1] .. list[b]
and call it `max`

b. compare list[a] and max

if (list[a] >= max)

the largest element in list[a]...list[b] is list[a]

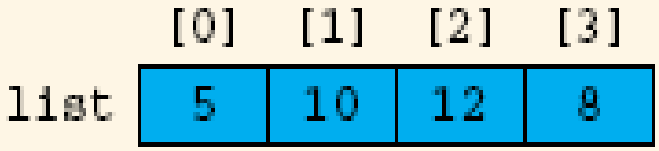
else

the largest element in list[a]...list[b] is max

Largest Value in Array

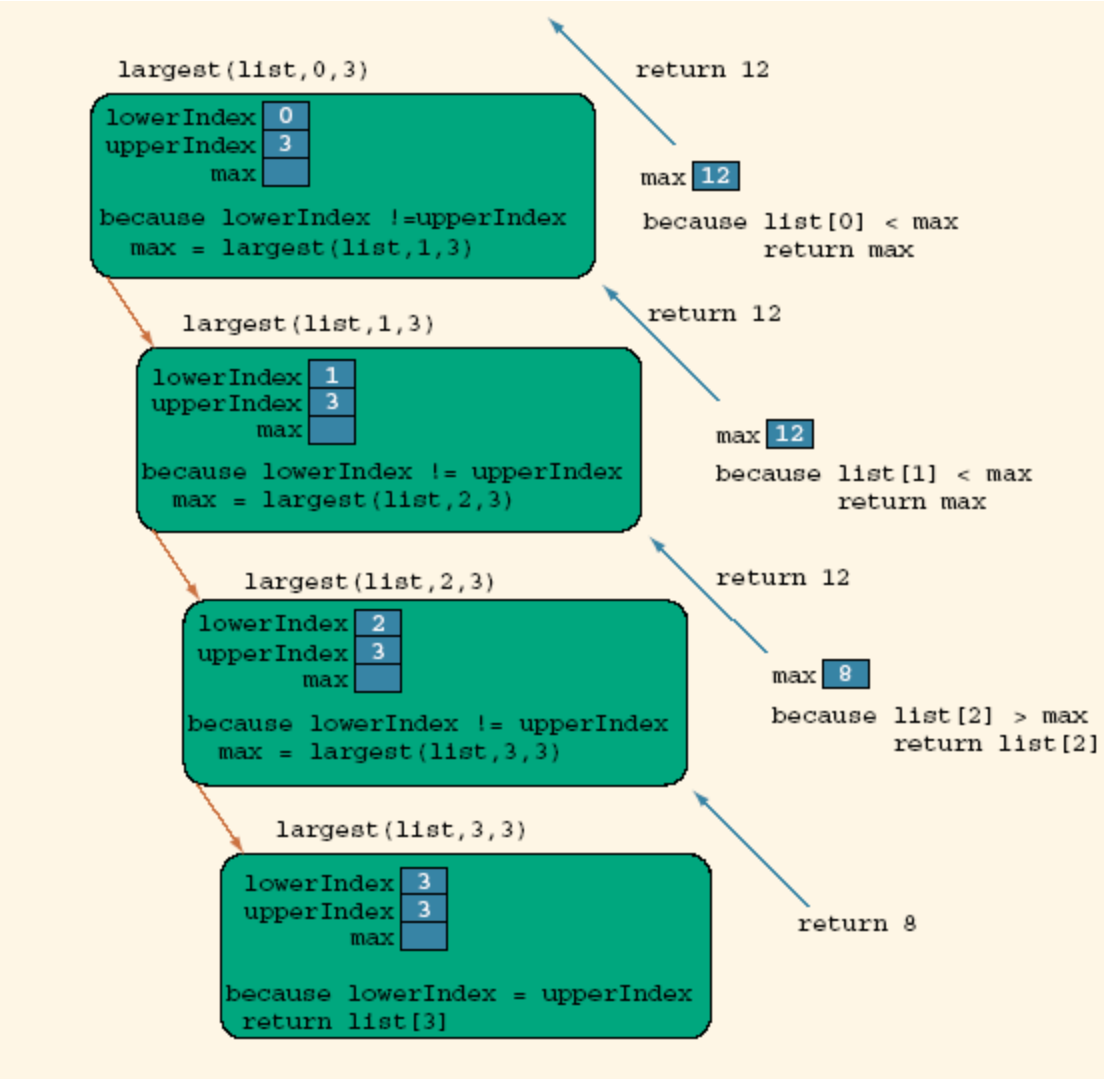
```
public static int largest(int[] list, int lowerIndex, int upperIndex) {
   int max;
   if (lowerIndex == upperIndex)
      return list[lowerIndex];
   else   {
      max = largest(list, lowerIndex + 1,
                    upperIndex);
      if (list[lowerIndex] >= max)
         return list[lowerIndex];
      else
         return max;
   }
}
```

Execution of `largest(list, 0, 3)`




```
System.out.println(largest(list, 0, 3));
```

Execution of `largest(list, 0, 3)`



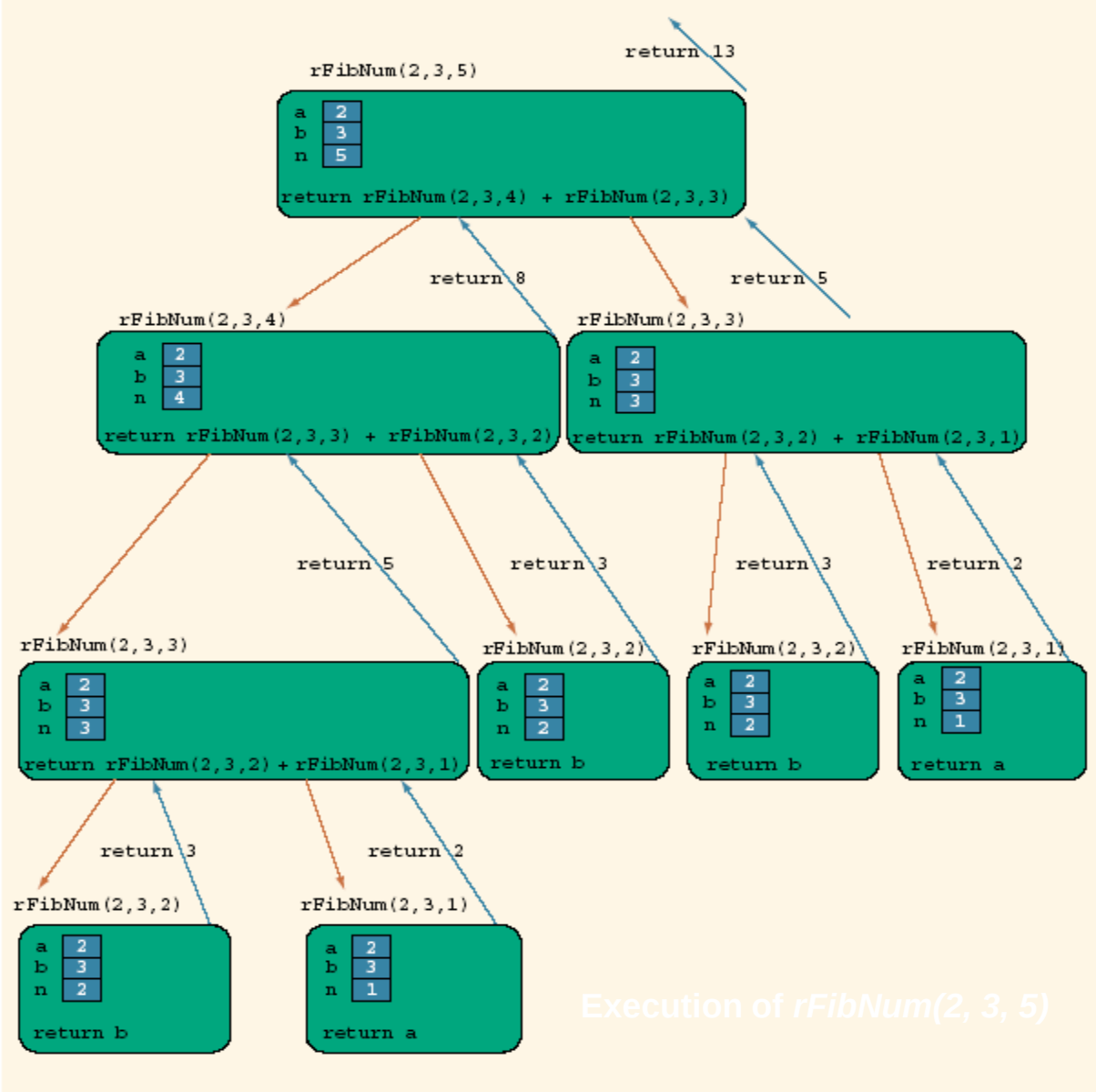
Recursive Fibonacci

$$rFibNum(a, b, n) = \begin{cases} a & \text{if } n = 1 \\ b & \text{if } n = 2 \\ rFibNum(a, b, n-1) + rFibNum(a, b, n-2) & \text{if } n > 2 \end{cases}$$

Recursive Fibonacci

```
public static int rFibNum(int a, int b, int n) {
    if (n == 1)
        return a;
    else if (n == 2)
        return b;
    else
        return rFibNum(a, b, n -1) +  rFibNum(a, b, n - 2);
}
```

Recursive Fibonacci



Execution of *rFibNum(2, 3, 5)*

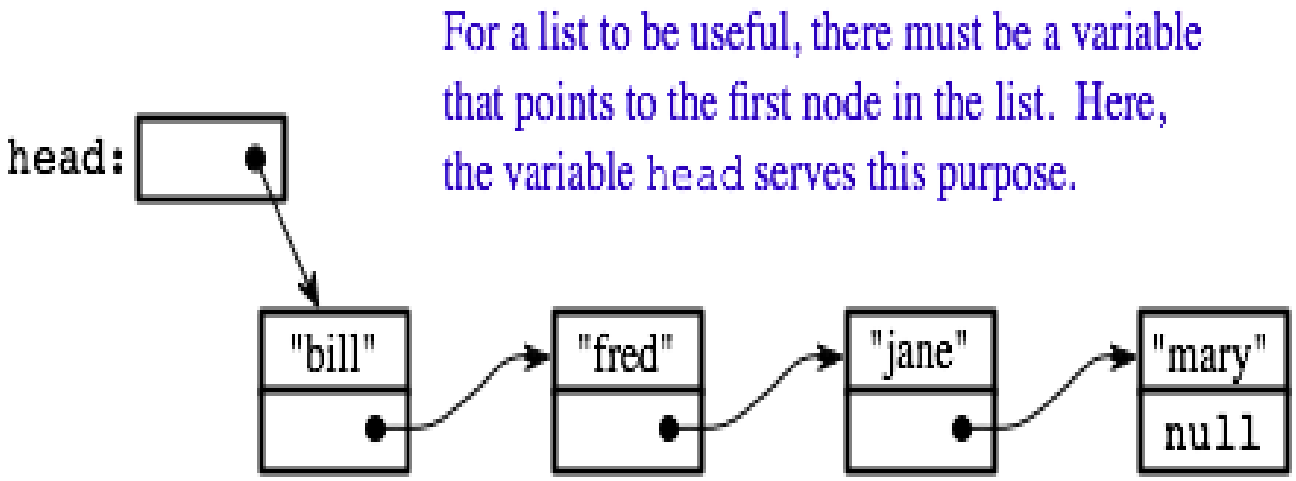
Recursion and the Method Call Stack

- 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

Recursive Data

- Also fundamental
- A linked list
- define a node

```
class Node {
    String item;
    Node next;
    .....
}
```




```
Node runner;    // A pointer that will be used to traverse the list.
runner = head;  // Start with runner pointing to the head of the list.
while ( runner != null ) {     // Continue until null is encountered.
   process( runner.item );     // Do something with the item in the current node.
   runner = runner.next;       // Move on to the next node in the list.
}
```