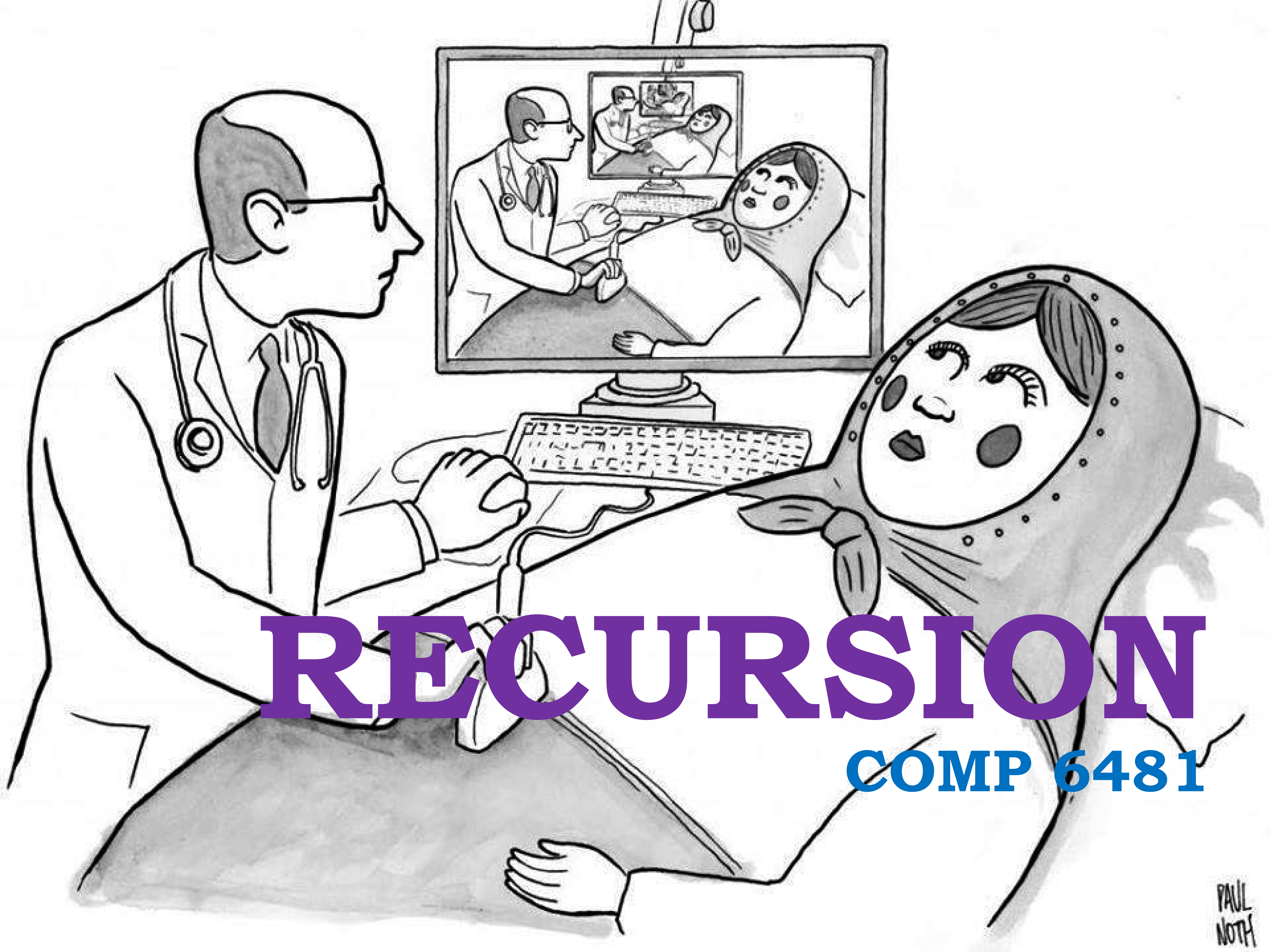
RECURSION

COMP 6481



IMAGINE YOU'RE IN A CUE OF PEOPLE

YOU COULD WALK UP TO THE FRONT AND COUNT ALL THE PEOPLE…

ASK THE PERSON IN FRONT OF YOU HOW MANY PEOPLE ARE IN FRONT OF THEM

THEY CAN ASK THE SAME QUESTION OF THE PERSON IN FRONT OF THEM

THEY CAN ASK THE SAME QUESTION OF THE PERSON IN FRONT OF THEM

THEY CAN ASK THE SAME QUESTION OF THE PERSON IN FRONT OF THEM

THE PERSON AT THE FRONT HAS NO ONE TO ASK…

THE FIRST PERSON IN LINE SAYS 0 PEOPLE ARE IN FRONT OF HIM

THE NEXT PERSON SAYS ONLY 1 PERSON IN FRONT OF ME

THE NEXT PERSON SAYS 2 PEOPLE IN FRONT OF HIM

ASK THE PERSON IN FRONT OF YOU HOW MANY PEOPLE ARE IN FRONT OF THEM

NOW YOU KNOW…

RECURSION — HOW IT WORKS?

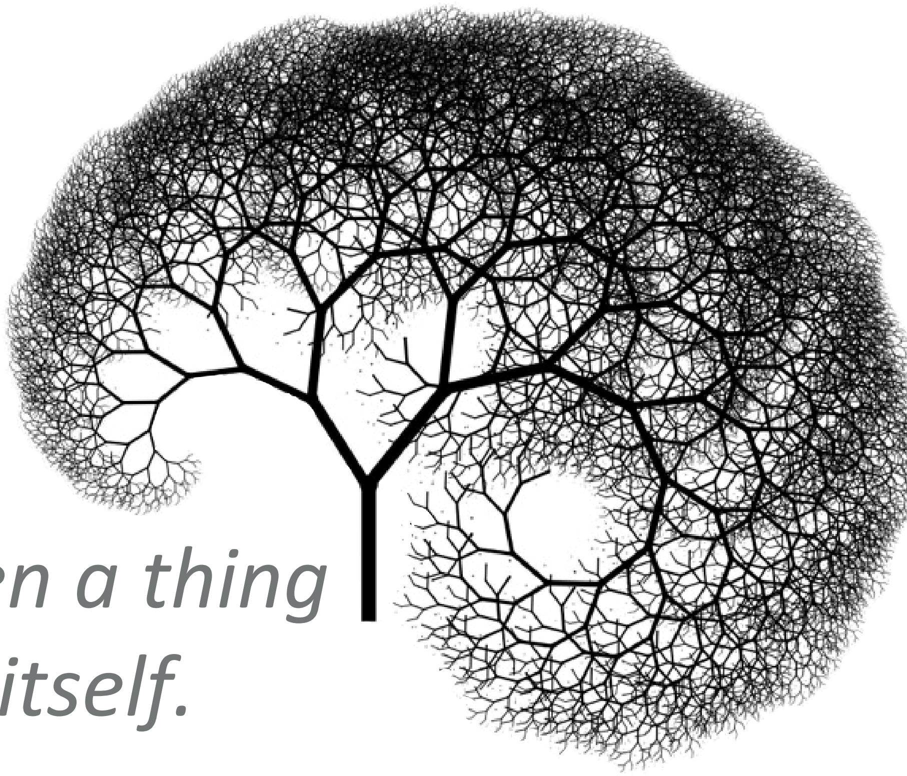
- Instead of solving the “hard” problem…
- Turn it into a slightly easier version of the hard problem.
- Eventually the problem will be so easy the answer is obvious.

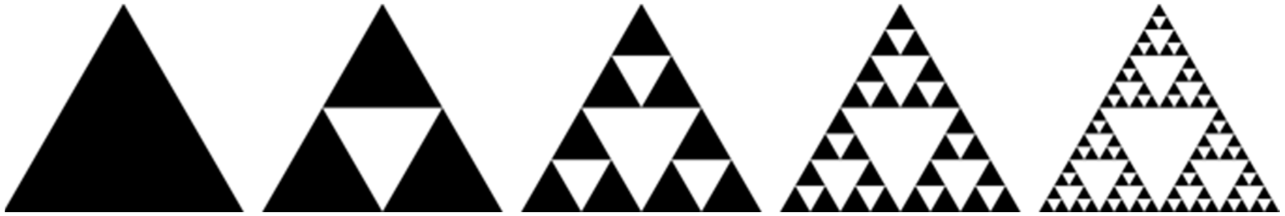
The base case

- Then work backwards to find the answer.

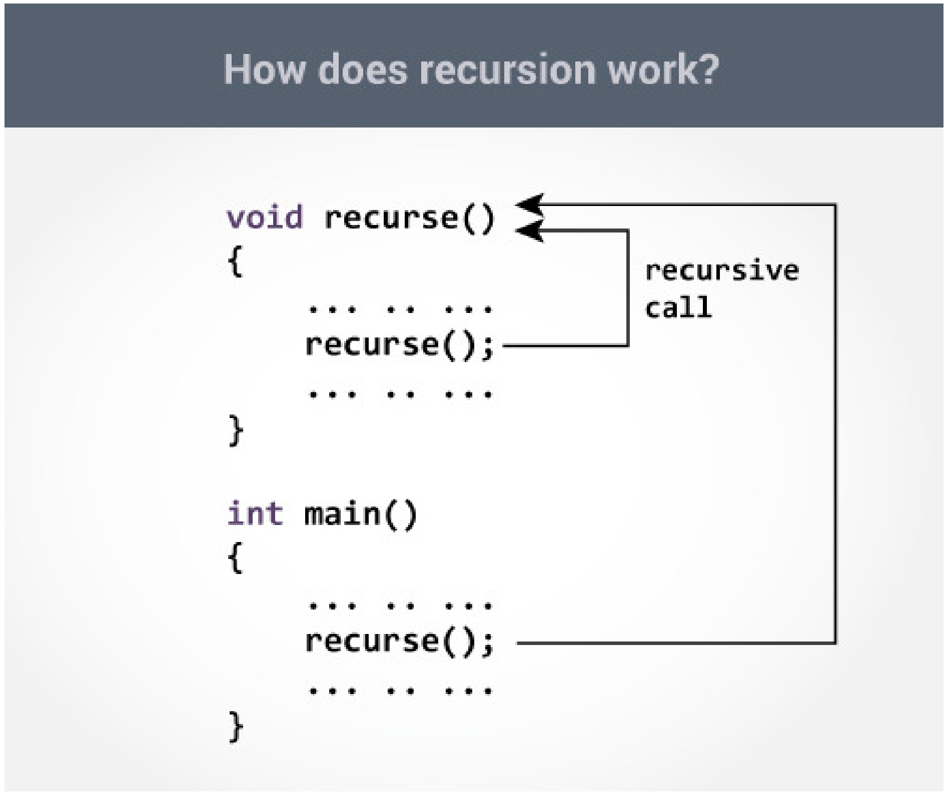
“

***Recursion** occurs when a thing is defined in terms of itself.*





VIDEO


How does recursion work?

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

https://www.youtube.com/watch?v=FyHloXKnPWc

RECURSIVE `VOID` METHODS

- A *recursive* method is a method that includes a call to itself
- Recursion is based on the general problem solving technique of breaking down a task into subtasks
  - In particular, recursion can be used whenever one subtask is a smaller version of the original task

EXAMPLE: VERTICAL NUMBERS

- The method `writeVertical` takes one (nonnegative) `int` argument, and writes that `int` with the digits going down the screen one per line.
- This task may be broken down into the following two subtasks
  - **Simple case:** If n<10, then write the number n to the screen
  - **Recursive Case:** If n>=10, then do two subtasks:
    - Output all the digits except the last digit
    - Output the last digit

VERTICAL NUMBERS

- Given the argument 1234, the output of the first subtask would be:

```
1
2
3
```

- The output of the second part would be:

```
4
```

VERTICAL NUMBERS

- The decomposition of tasks into subtasks can be used to derive the method definition:
    - Subtask 1 is a smaller version of the original task, so it can be implemented with a recursive call
    - Subtask 2 is just the simple case

ALGORITHM FOR VERTICAL NUMBERS

- Given parameter **`n`**:

```
if (n<10)
      System.out.println(n);
else {
      writeVertical(the number n with the last digit removed);
      System.out.println(the last digit of n);
    }
```

- Note: **`n/10`** is the number **`n`** with the last digit removed, and **`n%10`** is the last digit of **`n`**

A RECURSIVE **VOID** METHOD

Display 11.1 A Recursive void Method

```
public class RecursionDemo1
{
    public static void main(String[] args)
    {
        System.out.println("writeVertical(3):");
        writeVertical(3);

        System.out.println("writeVertical(12):");
        writeVertical(12);

        System.out.println("writeVertical(123):");
        writeVertical(123);
    }

    public static void writeVertical(int n)
    {
        if (n < 10)
        {
            System.out.println(n);
        }
        else //n is two or more digits long:
        {
            writeVertical(n/10);
            System.out.println(n%10);
        }
    }
}
```

Sample Dialogue

```
writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3
```

TRACING A RECURSIVE CALL

- Recursive methods are processed in the same way as any method call

  `writeVertical(123);`

  - When this call is executed, the argument **`123`** is substituted for the parameter **`n`**, and the body of the method is executed
  - Since **`123`** is not less than **`10`**, the **`else`** part is executed

TRACING A RECURSIVE CALL

- The else part begins with the method call:

```
writeVertical(n/10);
```

- Substituting `n` equal to `123` produces:

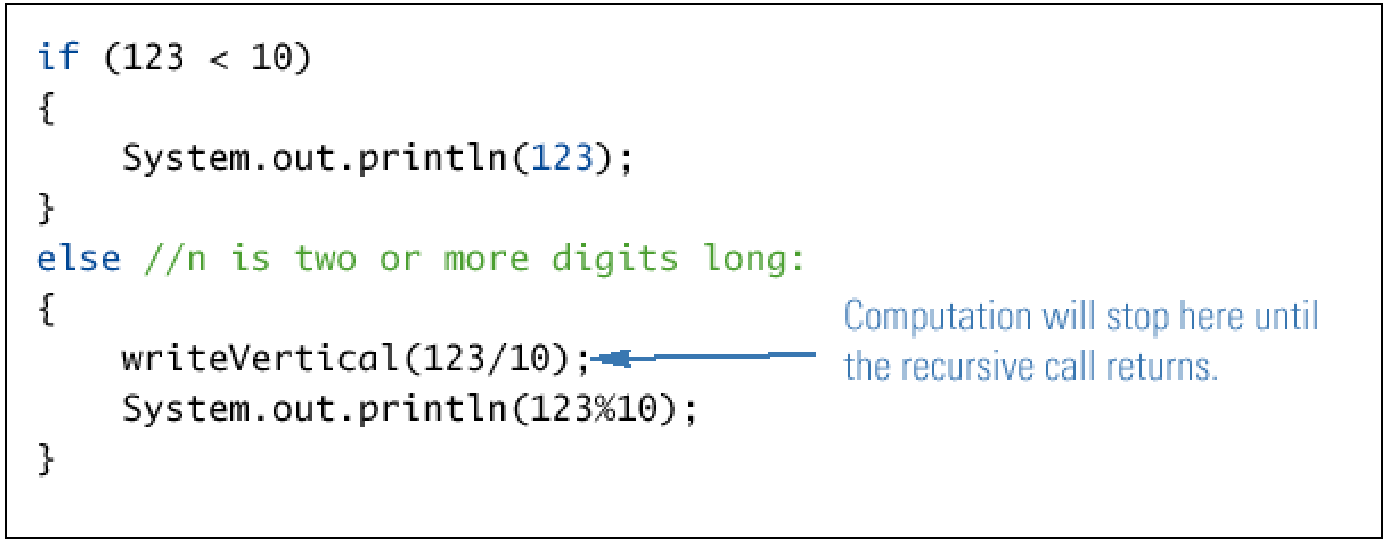
```
writeVertical(123/10);
```

- Which evaluates to

```
writeVertical(12);
```

- At this point, the current method computation is placed on hold, and the recursive call `writeVertical` is executed with the parameter `12`
- When the recursive call is finished, the execution of the suspended computation will return and continue from the point above

EXECUTION OF **WRITEVERTICAL(123)**



```
if (123 < 10)
{
    System.out.println(123);
}
else //n is two or more digits long:
{
    writeVertical(123/10);
    System.out.println(123%10);
}
```

Computation will stop here until the recursive call returns.

TRACING A RECURSIVE CALL

```
writeVertical(12);
```

- When this call is executed, the argument **`12`** is substituted for the parameter **`n`**, and the body of the method is executed
- Since **`12`** is not less than **`10`**, the **`else`** part is executed
- The else part begins with the method call:

```
writeVertical(n/10);
```

- Substituting **`n`** equal to **`12`** produces:

```
writeVertical(12/10);
```

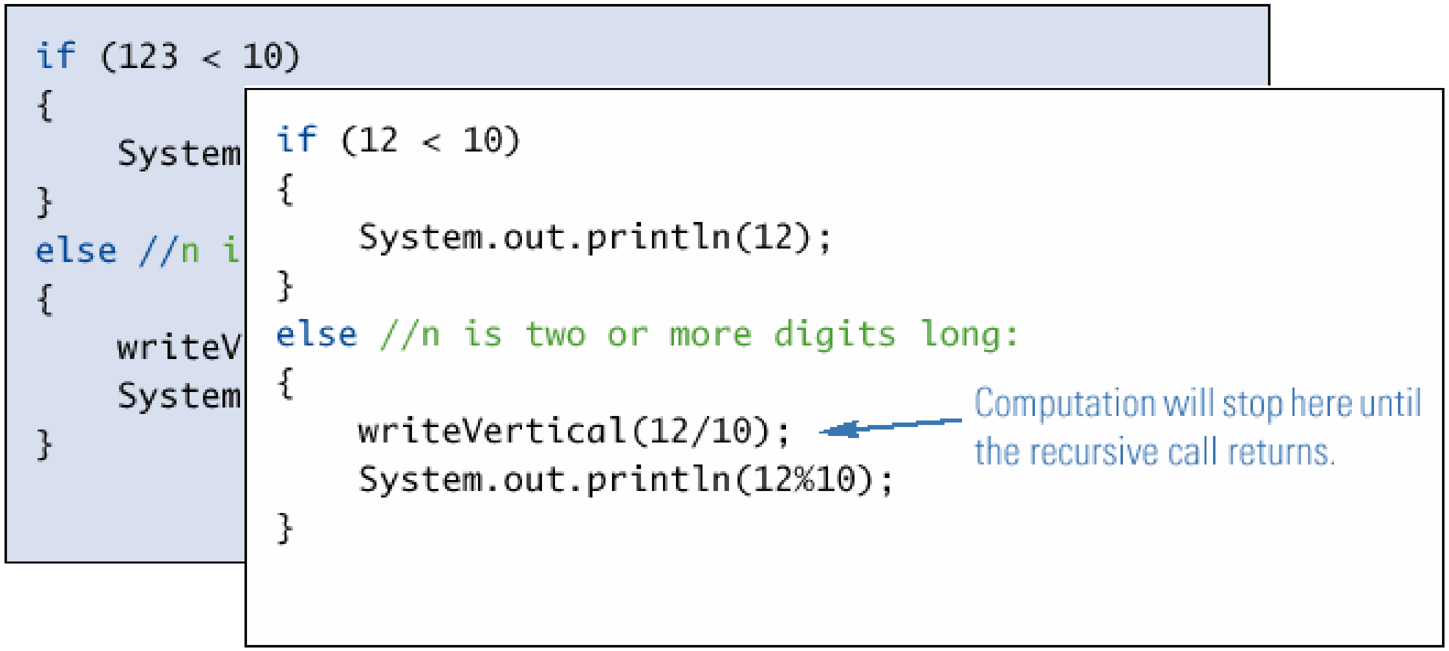
- Which evaluates to

```
write Vertical(1);
```

TRACING A RECURSIVE CALL

- So this second computation of `writeVertical` is suspended, leaving two computations waiting to resume , as the computer begins to execute another recursive call
- When this recursive call is finished, the execution of the second suspended computation will return and continue from the point above

EXECUTION OF **WRITEVERTICAL(12)**



```
if (123 < 10)
{
    System
}
else //n i
{
    writeV
    System
}
```

```
if (12 < 10)
{
    System.out.println(12);
}
else //n is two or more digits long:
{
    writeVertical(12/10);
    System.out.println(12%10);
}
```

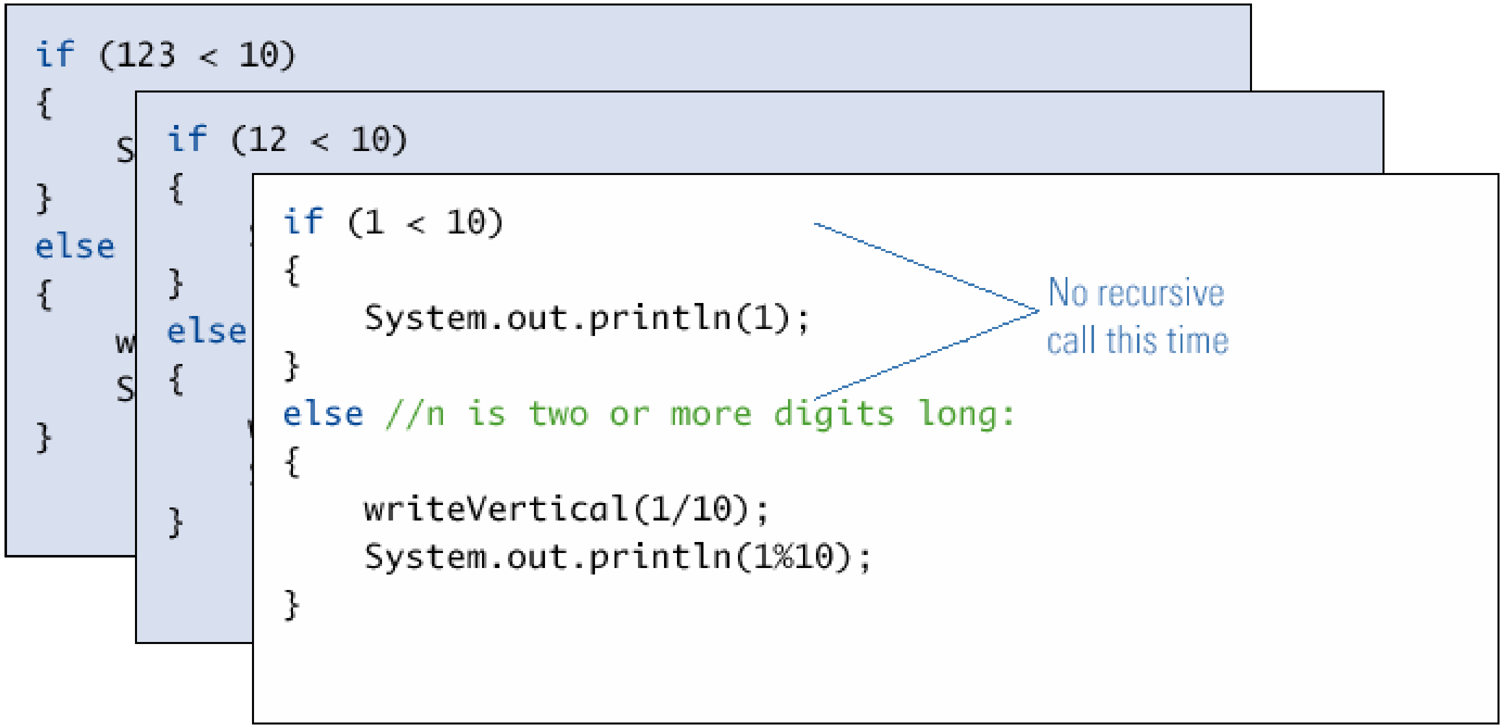
Computation will stop here until the recursive call returns.

TRACING A RECURSIVE CALL

`write Vertical(1);`

- When this call is executed, the argument **`1`** is substituted for the parameter **`n`**, and the body of the method is executed
- Since **`1`** is less than **`10`**, the **`if-else`** statement Boolean expression is finally true
- The output statement writes the argument **`1`** to the screen, and the method ends without making another recursive call
- Note that this is the stopping case

EXECUTION OF **`WRITEVERTICAL(1)`**



```
if (123 < 10)
{
    S
}
else
{
    w
    S
}
```

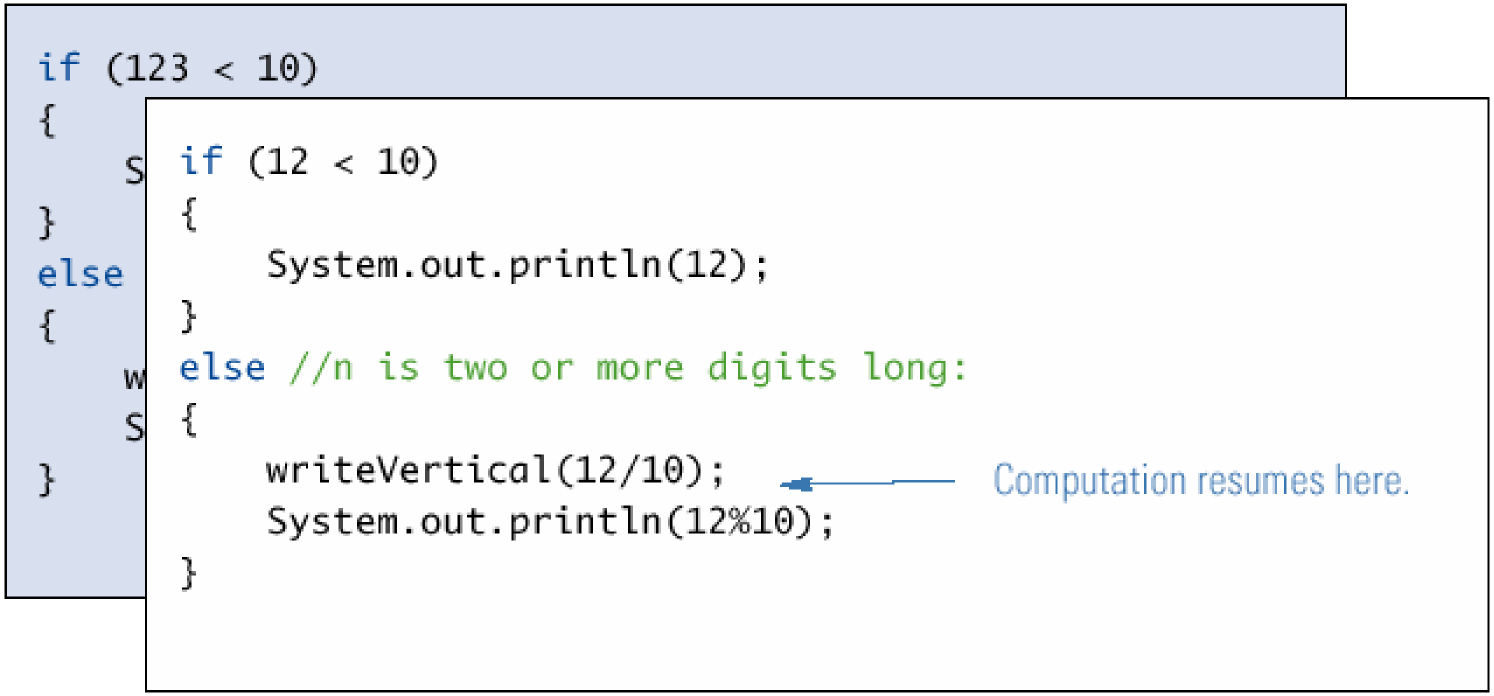
```
if (12 < 10)
{

}
else
{

}
```

```
if (1 < 10)
{
    System.out.println(1);
}
else //n is two or more digits long:
{
    writeVertical(1/10);
    System.out.println(1%10);
}
```

TRACING A RECURSIVE CALL

- When the call **`writeVertical(1)`** ends, the suspended computation that was waiting for it to end (the one that was initiated by the call **`writeVertical(12)`**) resumes execution where it left off
- It outputs the value **`12%10`**, which is **`2`**
- This ends the method
- Now the first suspended computation can resume execution

COMPLETION OF **`WRITEVERTICAL(12)`**



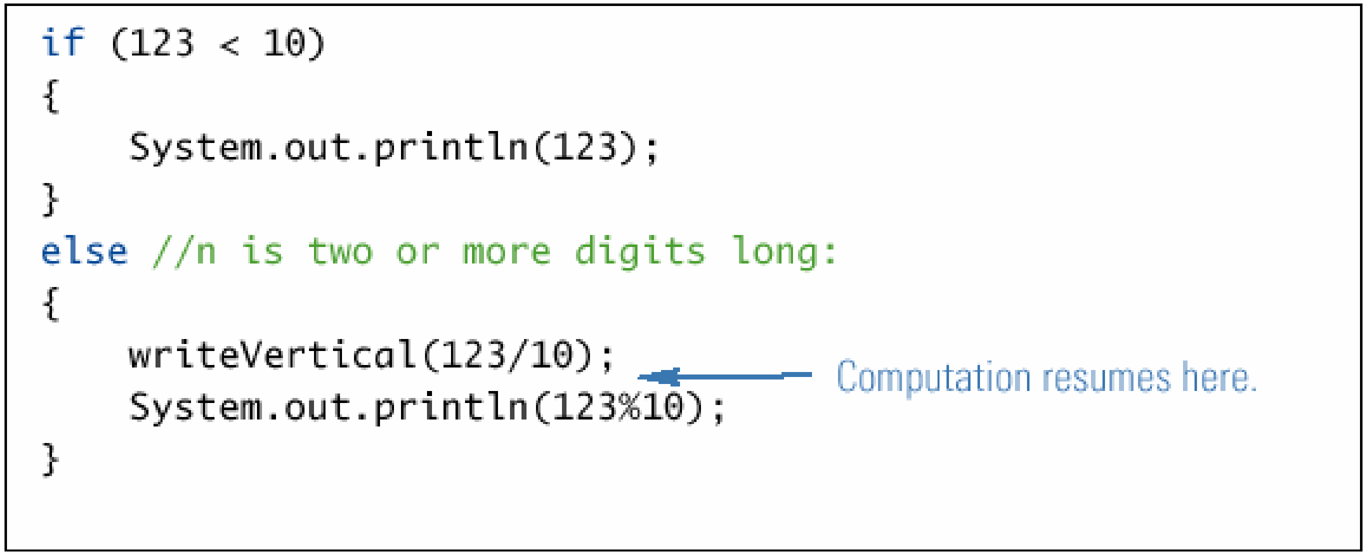
```
if (123 < 10)
{
     S
}
else
{
     w
     S
}
```

```
if (12 < 10)
{
     System.out.println(12);
}
else //n is two or more digits long:
{
     writeVertical(12/10);
     System.out.println(12%10);
}
```

TRACING A RECURSIVE CALL

- The first suspended method was the one that was initiated by the call **`writeVertical(123)`**
- It resumes execution where it left off
- It outputs the value **`123%10`**, which is **`3`**
- The execution of the original method call ends
- As a result, the digits 1,2, and 3 have been written to the screen one per line, in that order

COMPLETION OF **`WRITEVERTICAL(123)`**



```
if (123 < 10)
{
    System.out.println(123);
}
else //n is two or more digits long:
{
    writeVertical(123/10);
    System.out.println(123%10);
}
```

A CLOSER LOOK AT RECURSION

- The computer keeps track of recursive calls as follows:
  - When a method is called, the computer plugs in the arguments for the parameter(s), and starts executing the code
  - If it encounters a recursive call, it temporarily stops its computation
  - When the recursive call is completed, the computer returns to finish the outer computation

A CLOSER LOOK AT RECURSION

- When the computer encounters a recursive call, it must temporarily suspend its execution of a method
  - It does this because *it must know the result of the recursive call before it can proceed*
  - It saves all the information it needs to continue the computation later on, when it returns from the recursive call
- Ultimately, this entire process terminates when one of the recursive calls does not depend upon recursion to return

GENERAL FORM OF A RECURSIVE METHOD DEFINITION

- The general outline of a successful recursive method definition is as follows:
  - One or more cases that include one or more recursive calls to the method being defined
    - These recursive calls should solve "smaller" versions of the task performed by the method being defined
  - One or more cases that include no recursive calls: *base cases* or *stopping cases*

PITFALL: INFINITE RECURSION

- In the `writeVertical` example, the series of recursive calls eventually reached a call of the method that did not involve recursion (a stopping case)
- If, instead, every recursive call had produced another recursive call, then a call to that method would, in theory, run forever
  - This is called *infinite recursion*
  - In practice, such a method runs until the computer runs out of resources, and the program terminates abnormally

PITFALL: INFINITE RECURSION

- An alternative version of **`writeVertical`**
    - Note: No stopping case!

```
public static void newWriteVertical(int n)
{
   newWriteVertical(n/10);
   System.out.println(n%10);
}
```

PITFALL: INFINITE RECURSION

- A program with this method will compile and run
- Calling **`newWriteVertical(12)`** causes that execution to stop to execute the recursive call **`newWriteVertical(12/10)`**
    - Which is equivalent to **`newWriteVertical(1)`**
- Calling **`newWriteVertical(1)`** causes that execution to stop to execute the recursive call **`newWriteVertical(1/10)`**
    - Which is equivalent to **`newWriteVertical(0)`**

PITFALL: INFINITE RECURSION

- Calling **`newWriteVertical(0)`** causes that execution to stop to execute the recursive call **`newWriteVertical(0/10)`**
  - Which is equivalent to **`newWriteVertical(0)`**
  - . . . And so on, forever!
- Since the definition of **`newWriteVertical`** has no stopping case, the process will proceed *forever* (or until the computer runs out of resources)

STACKS FOR RECURSION

- To keep track of recursion (and other things), most computer systems use a *stack*
  - A stack is a very specialized kind of memory structure analogous to a stack of paper
  - As an analogy, there is also an inexhaustible supply of extra blank sheets of paper
  - Information is placed on the stack by writing on one of these sheets, and placing it on top of the stack (becoming the new top of the stack)
  - More information is placed on the stack by writing on another one of these sheets, placing it on top of the stack, and so on

STACKS FOR RECURSION

  - To get information out of the stack, the top paper can be read, *but only the top paper*
  - To get more information, the top paper can be thrown away, and then the new top paper can be read, and so on
- Since the last sheet put on the stack is the first sheet taken off the stack, a stack is called a *last-in/first-out* memory structure *(LIFO)*

STACKS FOR RECURSION

- To keep track of recursion, whenever a method is called, a new "sheet of paper" is taken
  - The method definition is copied onto this sheet, and the arguments are plugged in for the method parameters
  - The computer starts to execute the method body
  - When it encounters a recursive call, it stops the computation in order to make the recursive call
  - It writes information about the current method on the *sheet of paper*, and places it on the stack

STACKS FOR RECURSION

- A new *sheet of paper* is used for the recursive call
  - The computer writes a second copy of the method, plugs in the arguments, and starts to execute its body
  - When this copy gets to a recursive call, its information is saved on the stack also, and a new *sheet of paper* is used for the new recursive call

STACKS FOR RECURSION

- This process continues until some recursive call to the method completes its computation without producing any more recursive calls
  - Its *sheet of paper* is then discarded
- Then the computer goes to the top *sheet of paper* on the stack
  - This sheet contains the partially completed computation that is waiting for the recursive computation that just ended
  - Now it is possible to proceed with that suspended computation

STACKS FOR RECURSION

- After the suspended computation ends, the computer discards its corresponding sheet of paper (the one on top)
- The suspended computation that is below it on the stack now becomes the computation on top of the stack
- This process continues until the computation on the bottom sheet is completed

STACKS FOR RECURSION

- Depending on how many recursive calls are made, and how the method definition is written, the stack may grow and shrink in any fashion
- The stack of paper analogy has its counterpart in the computer
  - The contents of one of the *sheets of paper* is called a *stack frame* or *activation record*
  - The stack frames don't actually contain a complete copy of the method definition, but reference a single copy instead

PITFALL: STACK OVERFLOW

- There is always some limit to the size of the stack
  - If there is a long chain in which a method makes a call to itself, and that call makes another recursive call, . . . , and so forth, there will be many suspended computations placed on the stack
  - If there are too many, then the stack will attempt to grow beyond its limit, resulting in an error condition known as a *stack overflow*
- A common cause of stack overflow is infinite recursion

RECURSION VERSUS ITERATION

- Recursion is not absolutely necessary
  - Any task that can be done using recursion can also be done in a nonrecursive manner
  - A nonrecursive version of a method is called an *iterative version*
- An iteratively written method will typically use loops of some sort in place of recursion
- A recursively written method can be simpler, but will usually run slower and use more storage than an equivalent iterative version

ITERATIVE VERSION OF **WRITEVERTICAL**

Display 11.2 **Iterative Version of the Method in Display 11.1**

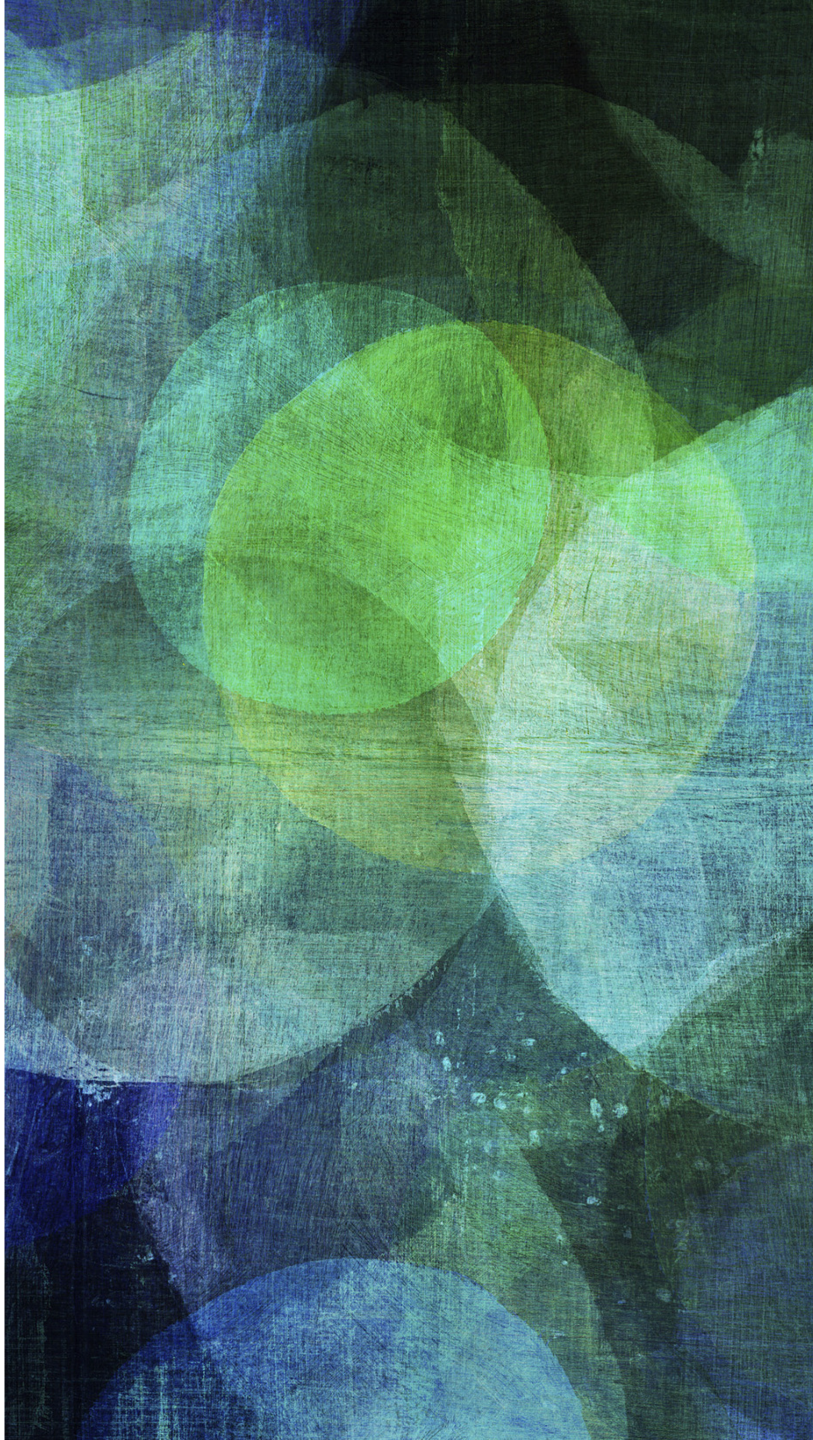
```
public static void writeVertical(int n)
{
    int nsTens = 1;
    int leftEndPiece = n;
    while (leftEndPiece > 9)
    {
        leftEndPiece = leftEndPiece/10;
        nsTens = nsTens*10;
    }
    //nsTens is a power of ten that has the same number
    //of digits as n. For example, if n is 2345, then
    //nsTens is 1000.

    for (int powerOf10 = nsTens;
            powerOf10 > 0; powerOf10 = powerOf10/10)
    {
        System.out.println(n/powerOf10);
        n = n%powerOf10;
    }
}
```

RECURSIVE METHODS THAT RETURN A VALUE

- Recursion is not limited to `void` methods
- A recursive method can return a value of any type
- An outline for a successful recursive method that returns a value is as follows:
  - One or more cases in which the value returned is computed in terms of calls to the same method
  - the arguments for the recursive calls should be intuitively "smaller"
  - One or more cases in which the value returned is computed without the use of any recursive calls (the base or stopping cases)

RECURSION

Continued



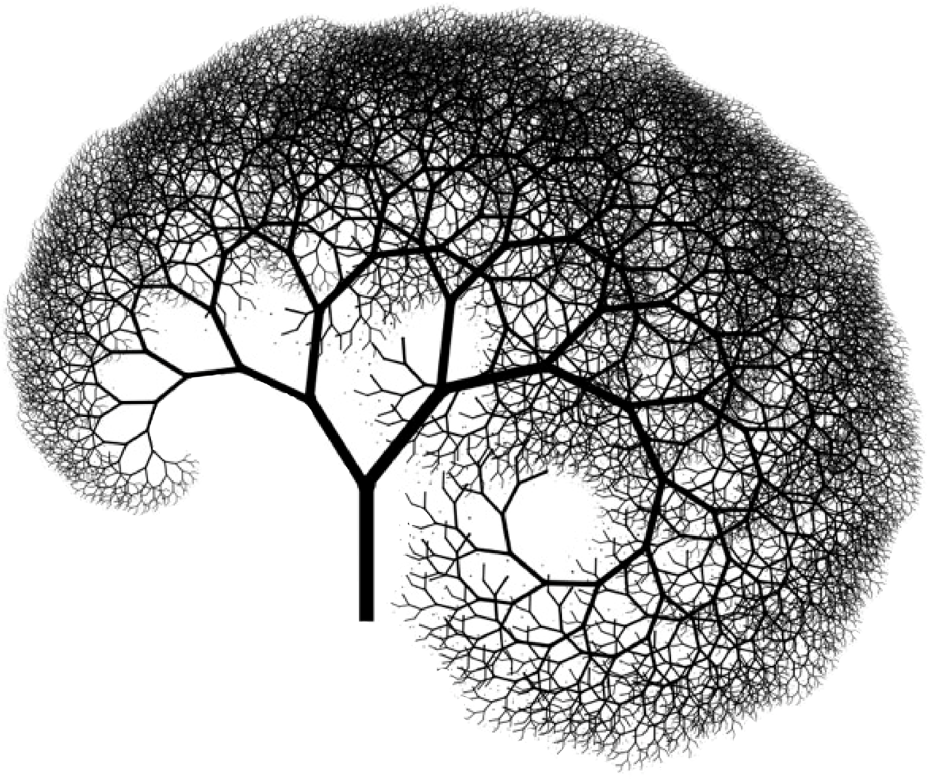
RECURSION — HOW IT WORKS?

- Instead of solving the “hard” problem…
- Turn it into a slightly easier version of the hard problem.
- Eventually the problem will be so easy the answer is obvious.

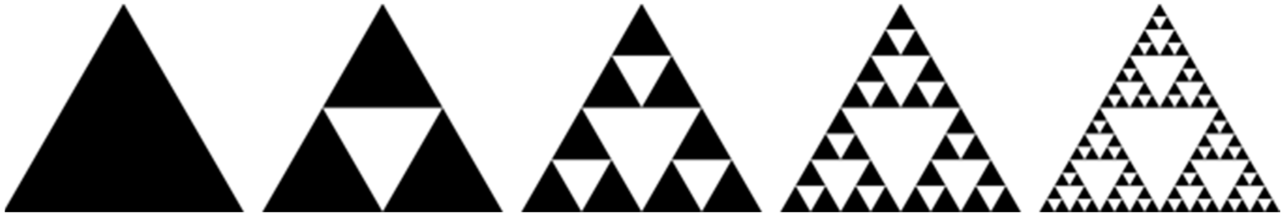
The base case

- Then work backwards to find the answer.

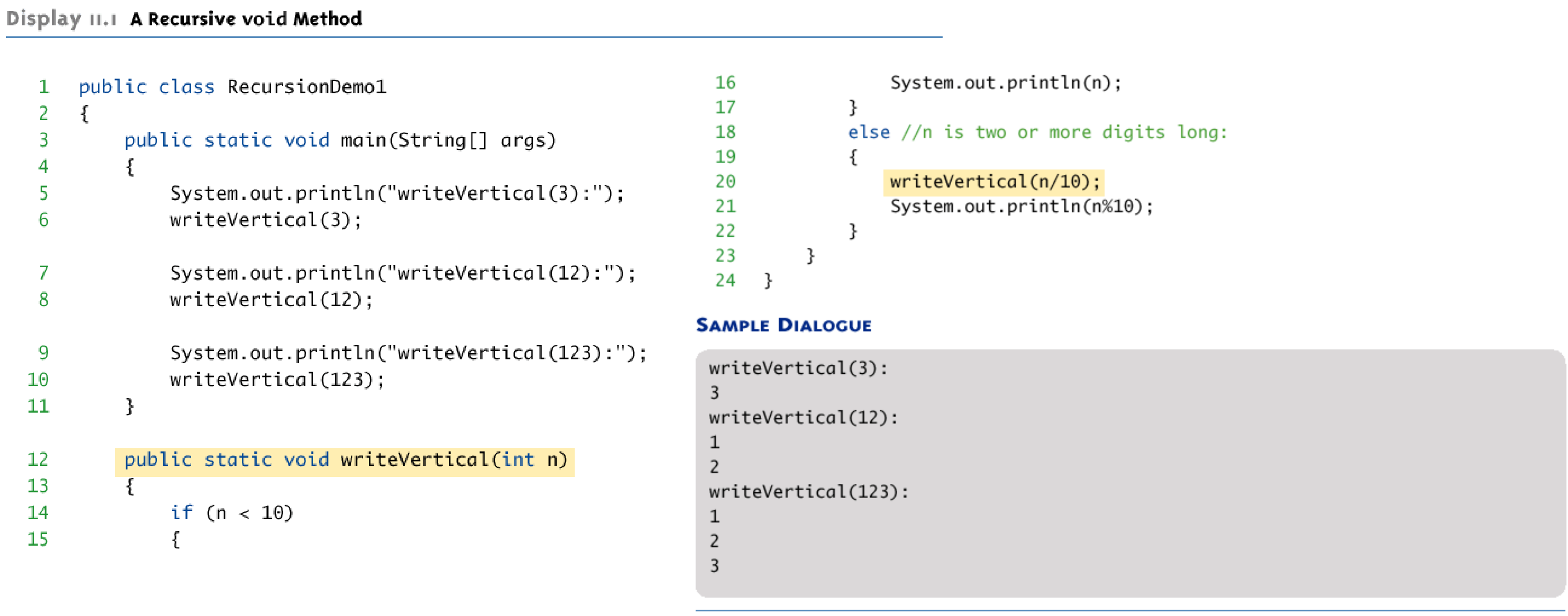
“



***Recursion** occurs when a thing is defined in terms of itself.*



AN EXAMPLE RECURSIVE **VOID** METHOD



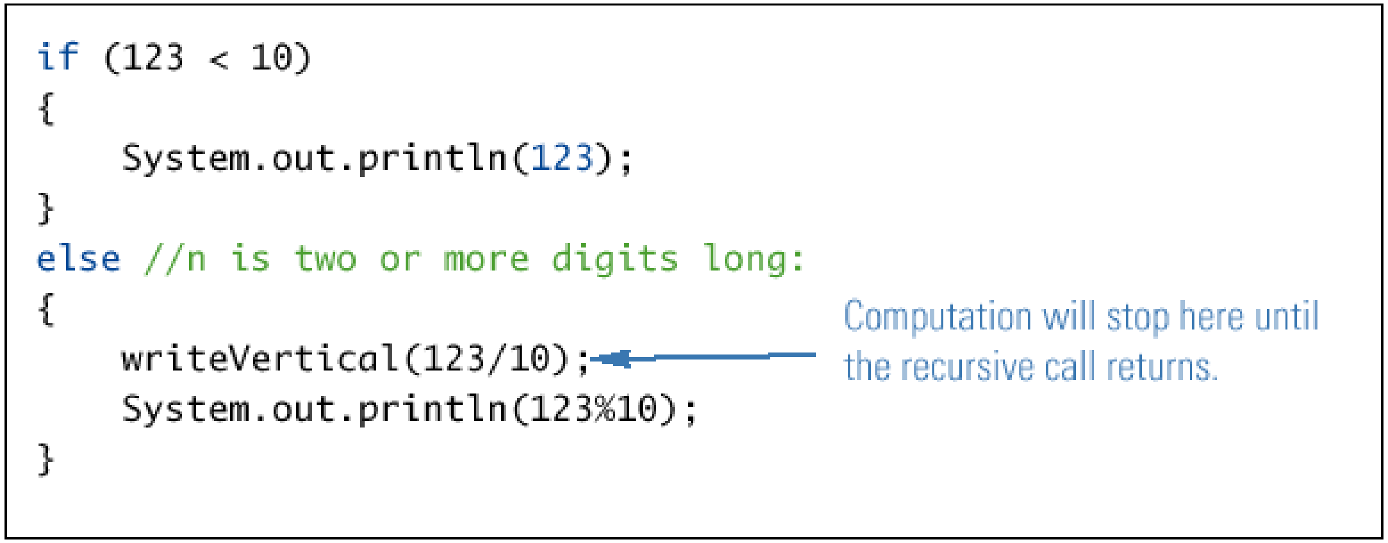
Display 11.1 A Recursive void Method

```java
public class RecursionDemo1
{
    public static void main(String[] args)
    {
        System.out.println("writeVertical(3):");
        writeVertical(3);

        System.out.println("writeVertical(12):");
        writeVertical(12);

        System.out.println("writeVertical(123):");
        writeVertical(123);
    }

    public static void writeVertical(int n)
    {
        if (n < 10)
        {
            System.out.println(n);
        }
        else //n is two or more digits long:
        {
            writeVertical(n/10);
            System.out.println(n%10);
        }
    }
}
```

SAMPLE DIALOGUE

```
writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3
```

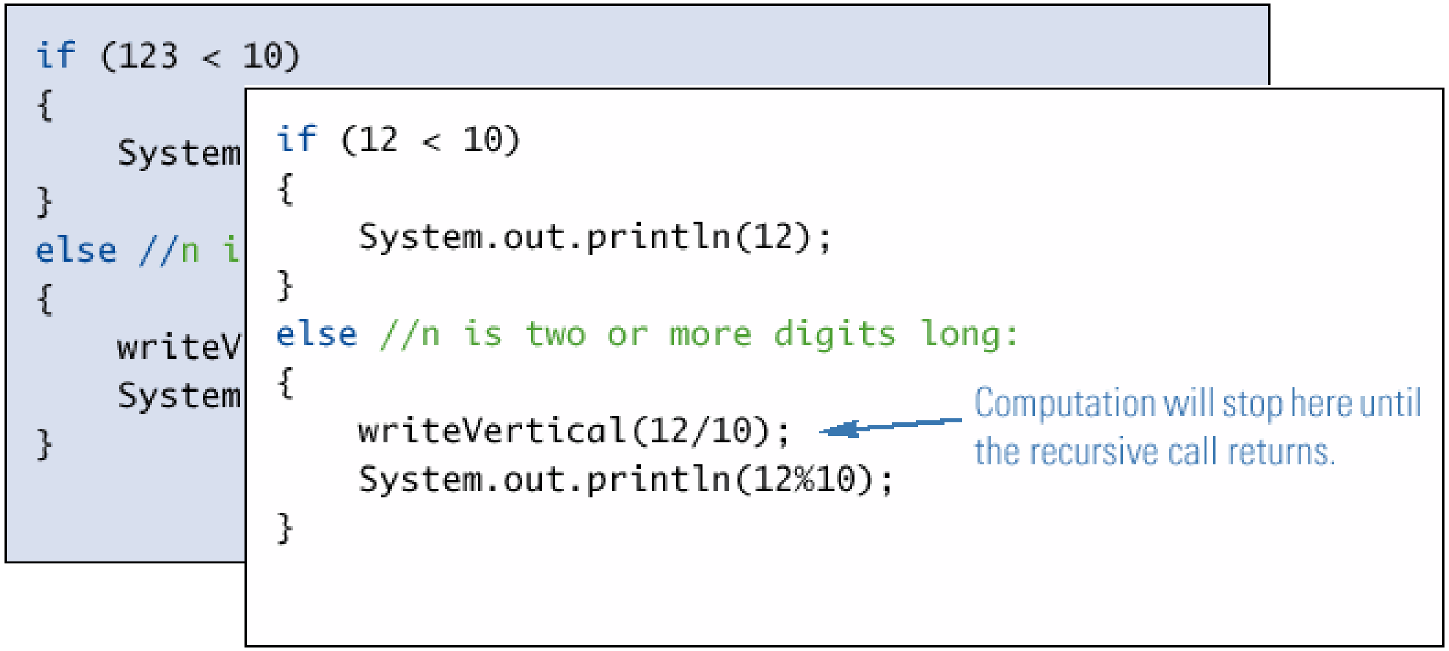
EXECUTION OF **WRITEVERTICAL(123)**



```
if (123 < 10)
{
    System.out.println(123);
}
else //n is two or more digits long:
{
    writeVertical(123/10);
    System.out.println(123%10);
}
```

Computation will stop here until the recursive call returns.

EXECUTION OF **WRITEVERTICAL(12)**

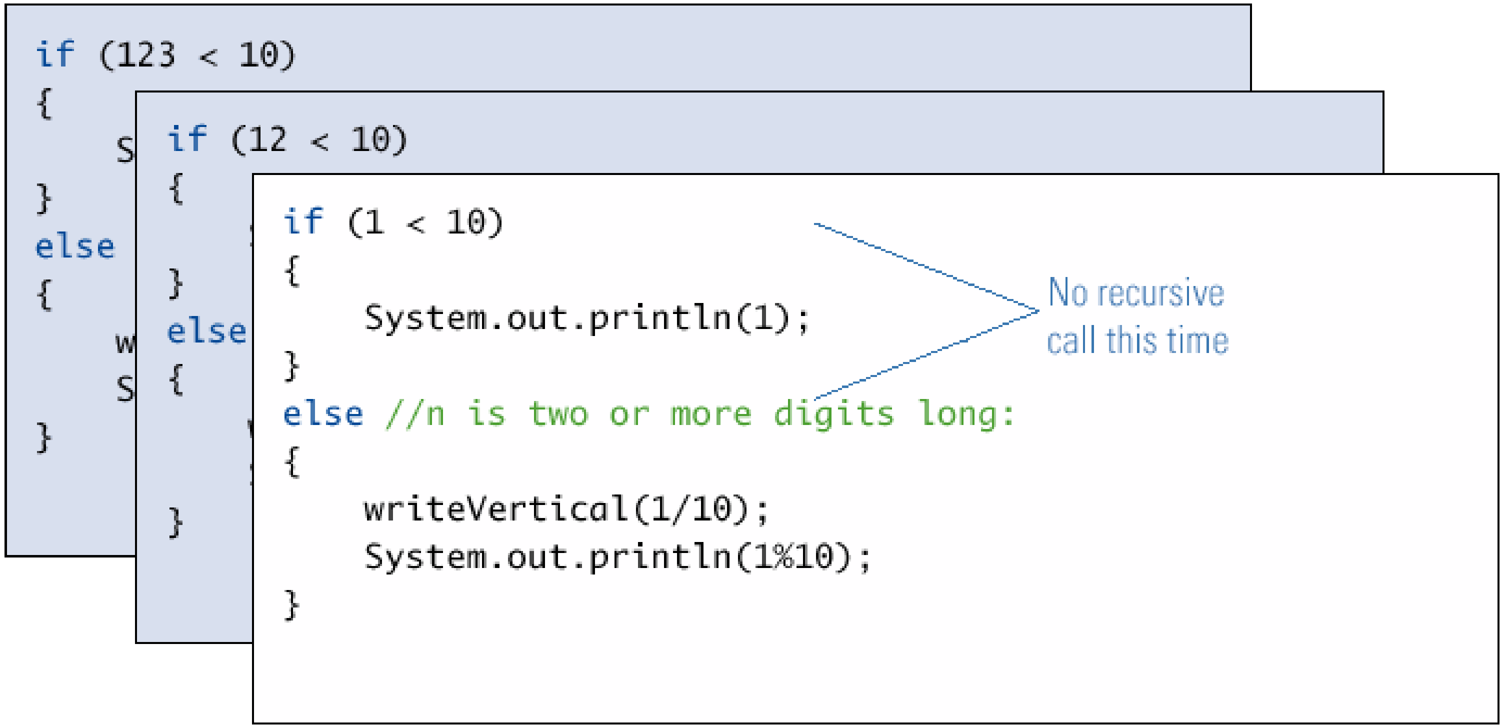


```
if (123 < 10)
{
    System
}
else //n i
{
    writeV
    System
}
```

```
if (12 < 10)
{
    System.out.println(12);
}
else //n is two or more digits long:
{
    writeVertical(12/10);
    System.out.println(12%10);
}
```

Computation will stop here until the recursive call returns.

EXECUTION OF **WRITEVERTICAL(1)**

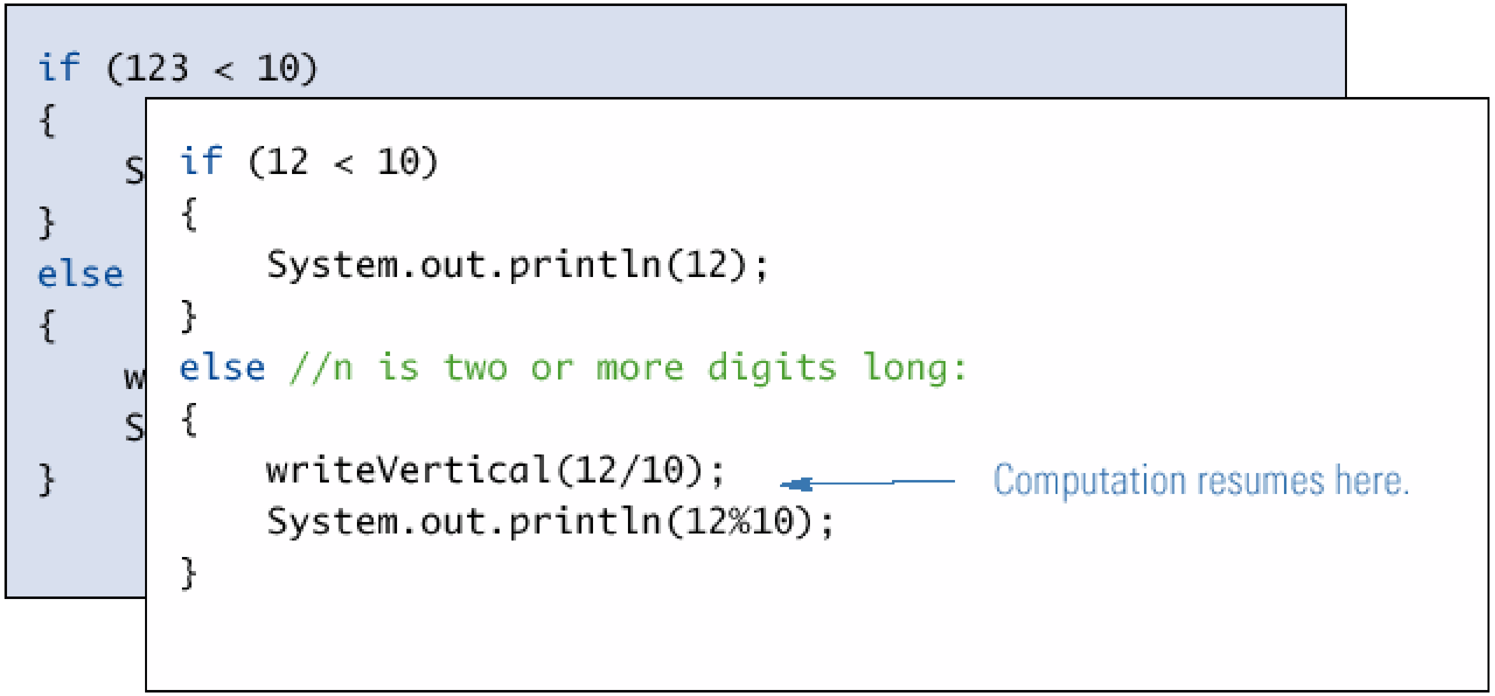


```
if (123 < 10)
{
    S
}
else
{
    w
    S
}
```

```
if (12 < 10)
{

}
else
{

}
```

```
if (1 < 10)
{
    System.out.println(1);
}
else //n is two or more digits long:
{
    writeVertical(1/10);
    System.out.println(1%10);
}
```

COMPLETION OF **WRITEVERTICAL(12)**

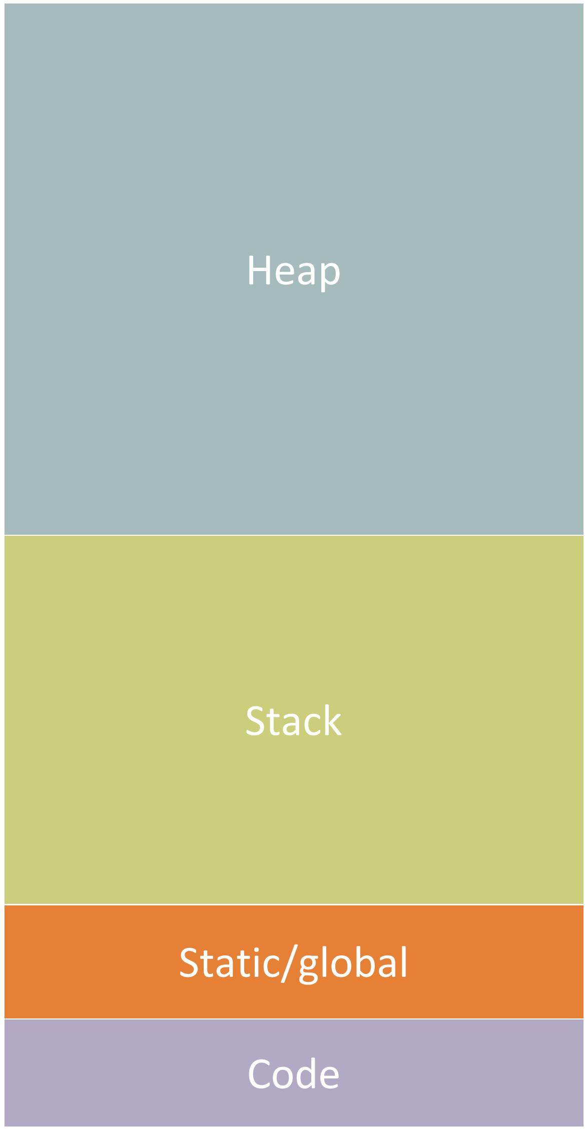


```
if (123 < 10)
{
     S
}
else
{
     w
     S
}
```

```
if (12 < 10)
{
     System.out.println(12);
}
else //n is two or more digits long:
{
     writeVertical(12/10);
     System.out.println(12%10);
}
```

STACK

- Regions of memory where data is added or removed in a last-in-first-out (LIFO) manner.
- Stores information about the active subroutines of a computer program



ANOTHER POWERS METHOD

- The method **`pow`** from the Math class computes powers
  - It takes two arguments of type **`double`** and returns a value of type **`double`**
- The recursive method **`power`** takes two arguments of type **`int`** and returns a value of type **`int`**
  - The definition of **`power`** is based on the following formula:

ANOTHER POWERS METHOD

- In terms of Java, the value returned by `power(x, n)` for `n>0` should be the same as

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

- When `n=0`, then `power(x, n)` should return `1`
  - This is the stopping case

THE RECURSIVE METHOD **POWER** (PART 1 OF 2)

Display 11.3 The Recursive Method power

```
public class RecursionDemo2
{
    public static void main(String[] args)
    {
        for (int n = 0; n < 4; n++)
            System.out.println("3 to the power " + n
                + " is " + power(3, n));
    }

    public static int power(int x, int n)
    {
        if (n < 0)
        {
            System.out.println("Illegal argument to power.");
            System.exit(0);
        }
```

THE RECURSIVE METHOD **POWER** (PART 1 OF 2)

Display 11.3 The Recursive Method power (continued)

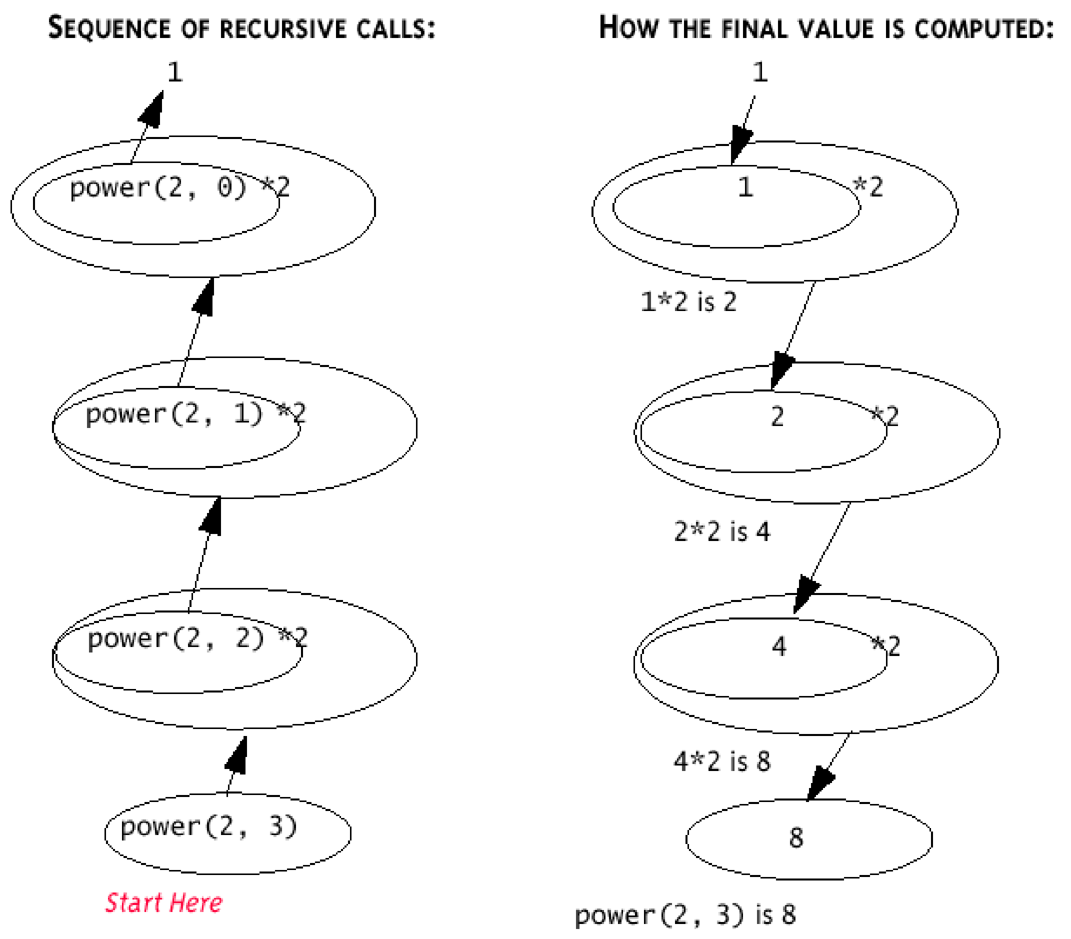
```
        if (n > 0)
            return ( power(x, n - 1)*x );
        else // n == 0
            return (1);
    }
}
```

Sample Dialogue

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

EVALUATING THE RECURSIVE METHOD CALL `POWER(2,3)`

Display 11.4 Evaluating the Recursive Method Call power(2,3)



THINKING RECURSIVELY

- If a problem lends itself to recursion, it is more important to think of it in recursive terms, rather than concentrating on the stack and the suspended computations

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

- In the case of methods that return a value, there are three properties that must be satisfied, as follows:

THINKING RECURSIVELY

1. There is no infinite recursion
   - Every chain of recursive calls must reach a stopping case
2. Each stopping case returns the correct value for that case
3. For the cases that involves recursion: *if* all recursive calls return the correct value, *then* the final value returned by the method is the correct value
   - These properties follow a technique also known as *mathematical induction*

TAIL RECURSION

- Tail recursion is a special case of recursion where the calling function does no more computation after making a recursive call. For example, the function

```
int f(int x, int y) {
  if (y == 0) {
    return x;
  }

  return f(x*y, y-1);
}
```

Tail recursive since the final instruction is a recursive call

```
int g(int x) {
  if (x == 1) {
    return 1;
  }

  int y = g(x-1);

  return x*y;
}
```

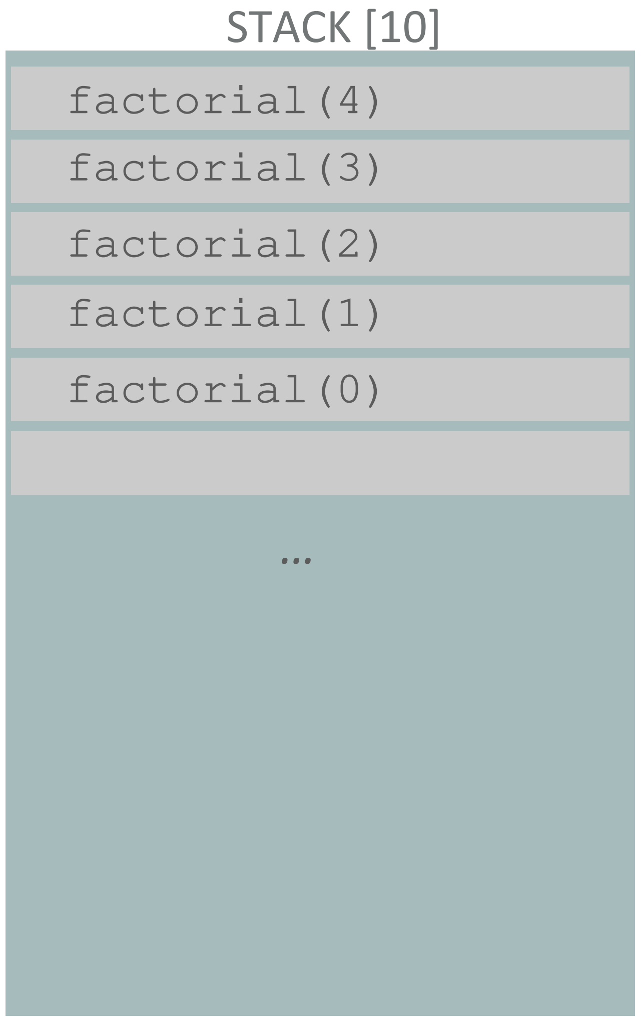
Not tail recursive since it does some computation after the recursive call has returned.

TAIL RECURSION VS GENERAL RECURSION EXAMPLE

```
int factorial(int n) {
  if (n == 0)
    return 1;

  return n*factorial(n-1);
}
```

TAIL RECURSION VS GENERAL RECURSION EXAMPLE

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



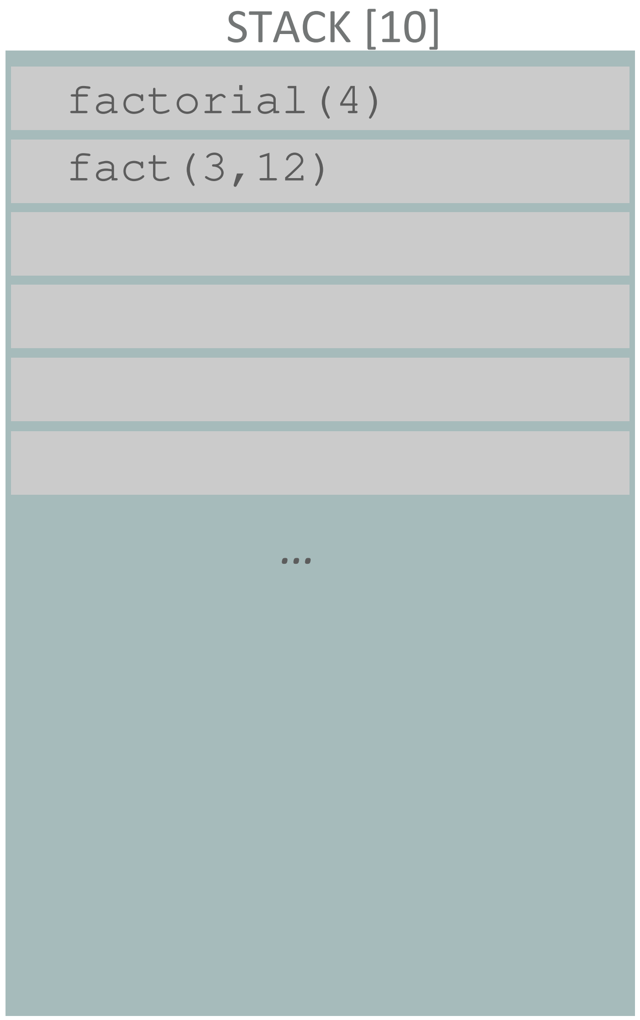
TAIL RECURSION VS GENERAL RECURSION EXAMPLE

```
int factorial(int n) {
  if (n == 0)
          return 1;
       else
          return fact(n-1,n);
}

int fact(int n, int result) {
  if (n == 0)
    return result;
    else
          return fact (n-1, n*result);
}
```

TAIL RECURSION VS GENERAL RECURSION EXAMPLE

```
int factorial(int n) {
  return fact(n-1, n);
}

int fact(int n, int result) {
  if (n == 0)
    return 1;
    fact (n-1, n*result);
}
```

No pending computation!
Existing stack frame thrown out &
next recursion takes its place.



TAIL RECURSION

- When the recursive method does nothing after the recursive call except return the value then the method is called ***tail recursive***
- Tail recursive methods can be implemented more efficiently then general recursive methods.
- Easily be converted into an equivalent iterative algorithm
  - Your compiler may do this automatically for greater efficiency

RECURSIVE DESIGN TECHNIQUES

- The same rules can be applied to a recursive `void` method:
  1. There is no infinite recursion
  2. Each stopping case performs the correct action for that case
  3. For each of the cases that involve recursion: if all recursive calls perform their actions correctly, then the entire case performs correctly

BINARY SEARCH

- Problem: A store clerk needs to search a large list of numbers for credit cards that are no longer valid.

BINARY SEARCH

- Binary search uses a recursive method to search an array to find a specified value
- The array must be a sorted array:

  `a[0]≤a[1]≤a[2]≤. . . ≤  a[finalIndex]`
- If the value is found, its index is returned
- If the value is not found, -1 is returned
- Note: Each execution of the recursive method reduces the search space by about a half

BINARY SEARCH

- An algorithm to solve this task looks at the middle of the array or array segment first
- If the value looked for is smaller than the value in the middle of the array
  - Then the second half of the array or array segment can be ignored
  - This strategy is then applied to the first half of the array or array segment

BINARY SEARCH

- If the value looked for is larger than the value in the middle of the array or array segment
  - Then the first half of the array or array segment can be ignored
  - This strategy is then applied to the second half of the array or array segment
- If the value looked for is at the middle of the array or array segment, then it has been found
- If the entire array (or array segment) has been searched in this way without finding the value, then it is not in the array

PSEUDOCODE FOR BINARY SEARCH

Display 11.5 Pseudocode for Binary Search

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
/**
 Precondition:
 a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
*/
```

TO LOCATE THE VALUE KEY:

```
if (first > last) //A stopping case
    return -1;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
        return mid;
    else if key < a[mid] //A case with recursion
        return the result of searching a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        return the result of searching a[mid + 1] through a[last];
 }
```

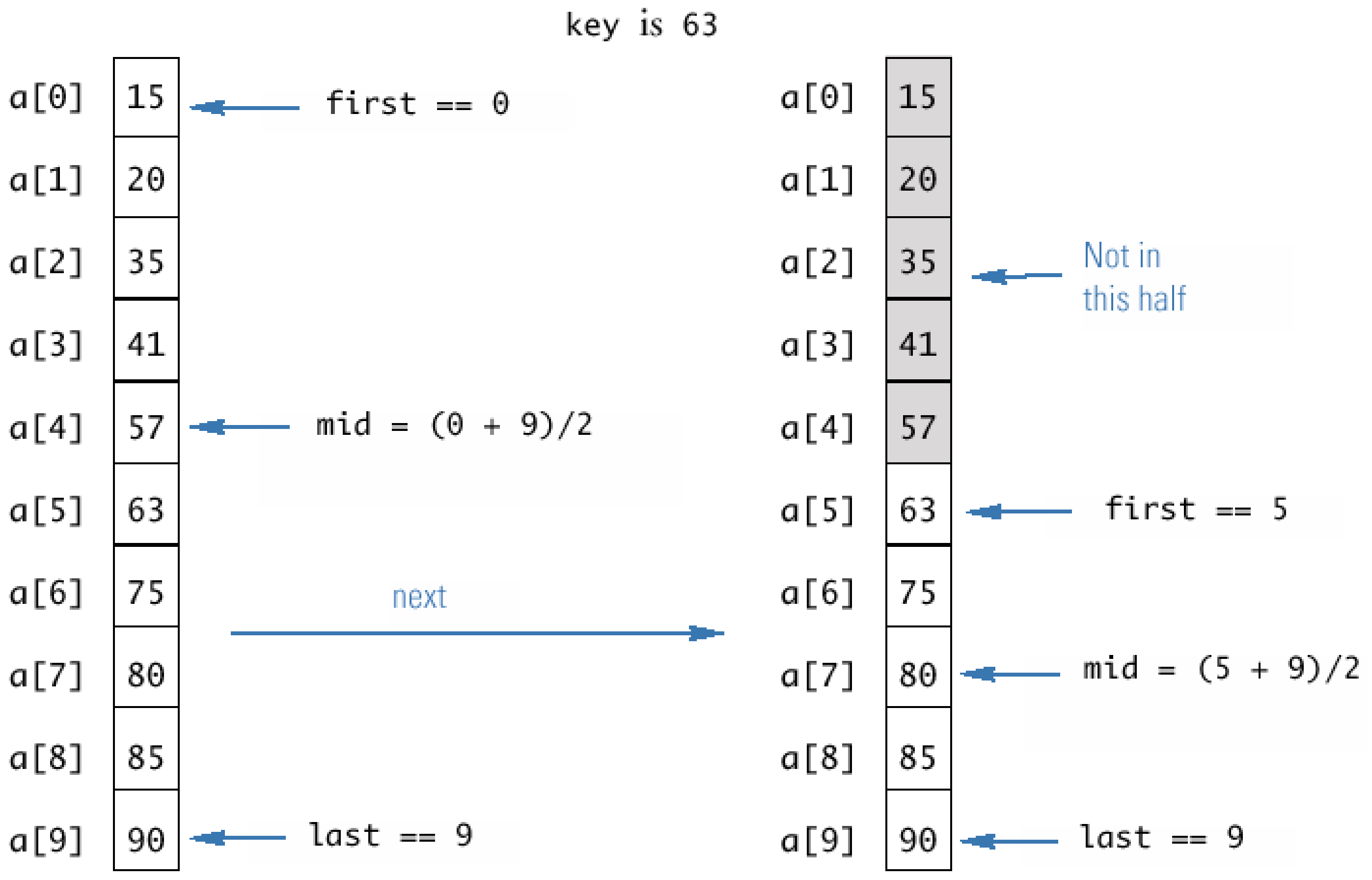
RECURSIVE METHOD FOR BINARY SEARCH

Display 11.6 Recursive Method for Binary Search

```java
public class BinarySearch
{
    /**
     Searches the array a for key. If key is not in the array segment, then −1 is
     returned. Otherwise returns an index in the segment such that key == a[index].
     Precondition: a[first] <= a[first + 1]<= ... <= a[last]
    */
    public static int search(int[] a, int first, int last, int key)
    {
        int result = 0; //to keep the compiler happy.

        if (first > last)
            result = −1;
        else
        {
            int mid = (first + last)/2;

            if (key == a[mid])
                result = mid;
            else if (key < a[mid])
                result = search(a, first, mid − 1, key);
            else if (key > a[mid])
                result = search(a, mid + 1, last, key);
        }
        return result;
    }
}
```

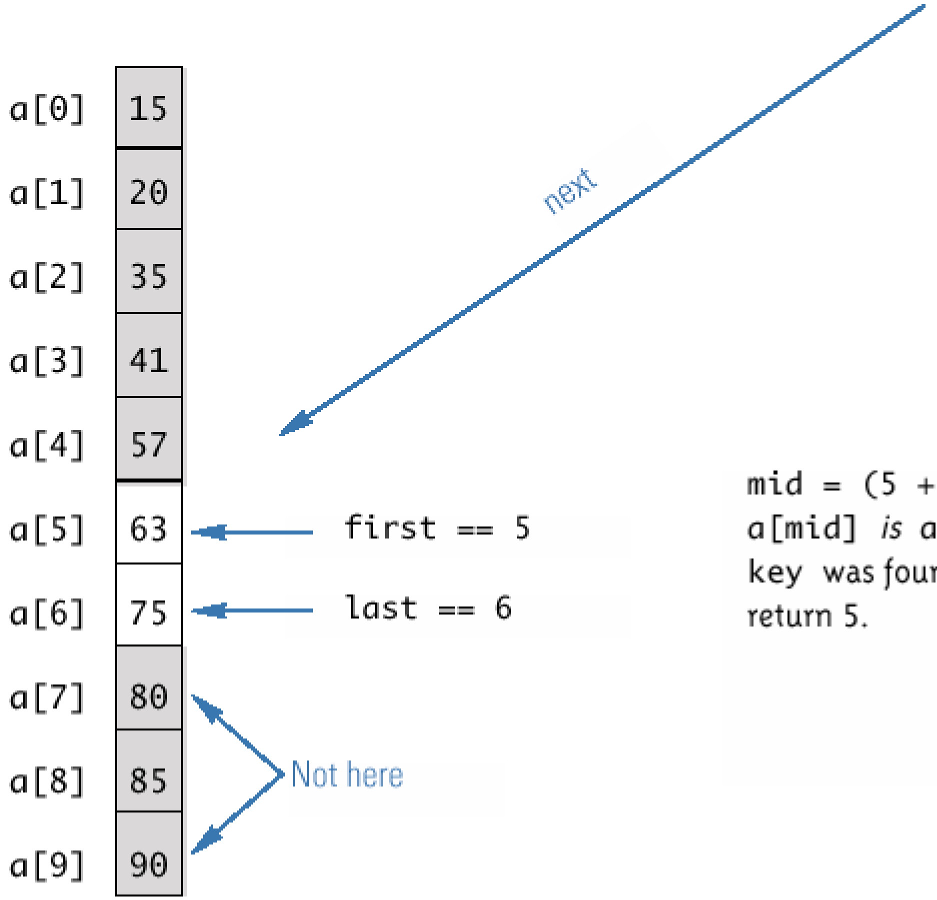
EXECUTION OF THE METHOD **SEARCH** (PART 1 OF 2)

Display 11.7 **Execution of the Method search**



EXECUTION OF THE METHOD **SEARCH** (PART 2 OF 2)

Display 11.7 Execution of the Method search (continued)



mid = (5 + 6)/2 *which is* 5
a[mid] *is* a[5] == 63
key was found.
return 5.

CHECKING THE `SEARCH` METHOD

1. There is no infinite recursion
   - On each recursive call, the value of `first` is increased, or the value of `last` is decreased
   - If the chain of recursive calls does not end in some other way, then eventually the method will be called with `first` larger than `last`

CHECKING THE **`SEARCH`** METHOD

2. Each stopping case performs the correct action for that case

- If **`first > last`**, there are no array elements between **`a[first]`** and **`a[last]`**, so **`key`** is not in this segment of the array, and **`result`** is correctly set to **`-1`**
- If **`key == a[mid]`**, **`result`** is correctly set to **`mid`**

CHECKING THE **`SEARCH`** METHOD

3. For each of the cases that involve recursion, *if* all recursive calls perform their actions correctly, *then* the entire case performs correctly

- If **`key < a[mid]`**, then **`key`** must be one of the elements **`a[first]`** through **`a[mid-1]`**, or it is not in the array
- The method should then search only those elements, which it does
- The recursive call is correct, therefore the entire action is correct

CHECKING THE `SEARCH` METHOD

- If `key > a[mid]`, then `key` must be one of the elements `a[mid+1]` through `a[last]`, or it is not in the array
- The method should then search only those elements, which it does
- The recursive call is correct, therefore the entire action is correct

The method `search` passes all three tests:

Therefore, it is a good recursive method definition

EFFICIENCY OF BINARY SEARCH

- The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order
  - About half the array is eliminated from consideration right at the start
  - Then a quarter of the array, then an eighth of the array, and so forth

EFFICIENCY OF BINARY SEARCH

- Given an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value, as compared to an average of 500 for a serial search algorithm
- The binary search algorithm has a worst-case running time that is logarithmic: $O(\log_2 n)$
  - A serial search algorithm is linear: $O(n)$
- If desired, the recursive version of the method **`search`** can be converted to an iterative version that will run more efficiently

ITERATIVE VERSION OF BINARY SEARCH (PART 1 OF 2)

Display 11.9 Iterative Version of Binary Search

```
/**
 Searches the array a for key. If key is not in the array segment, then −1 is
 returned. Otherwise returns an index in the segment such that key == a[index].
 Precondition: a[lowEnd] <= a[lowEnd + 1]<= ... <= a[highEnd]
*/
public static int search(int[] a, int lowEnd, int highEnd, int key)
{
    int first = lowEnd;
    int last = highEnd;
    int mid;

    boolean found = false; //so far
    int result = 0; //to keep compiler happy

    while ( (first <= last) && !(found) )
    {
        mid = (first + last)/2;
```

ITERATIVE VERSION OF BINARY SEARCH (PART 2 OF 2)

Display 11.9 Iterative Version of Binary Search (continued)

```
        if (key == a[mid])
        {
            found = true;
            result = mid;
        }
        else if (key < a[mid])
        {
            last = mid - 1;
        }
        else if (key > a[mid])
        {
            first = mid + 1;
        }
    }

    if (first > last)
        result = -1;

    return result;
}
```