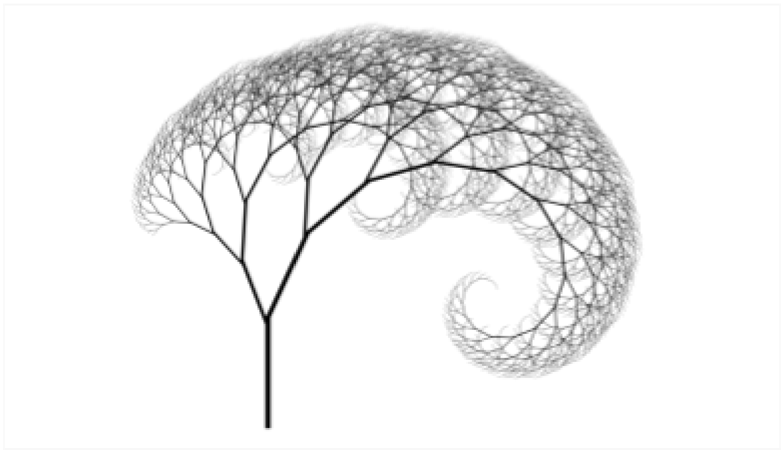
JAVA TIP NO.

github.com/Teabeans/Java_Learn/

UWB_TADEMO_001.pdf

- RECURSION -



Prepared For:

The University of Washington Bothell

CSS TA Admissions Committee

2019.08

By:

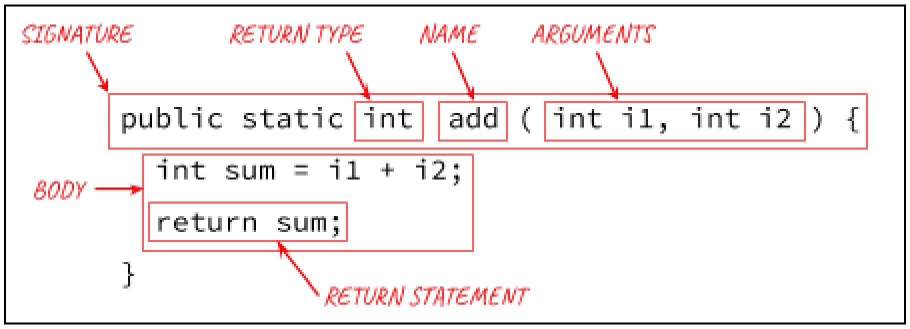
Tim Lum

twhlum@gmail.com

2019.07.18

0 - Prerequisites:

Method Definitions:
Sometimes also called a 'function', methods are a reusable set of procedures similar to a recipe or instruction manual. They are comprised of a **signature** (sometimes called a 'header' defining the name, inputs, outputs, and some other details of the method) and a **body** (where the actual instructions reside).

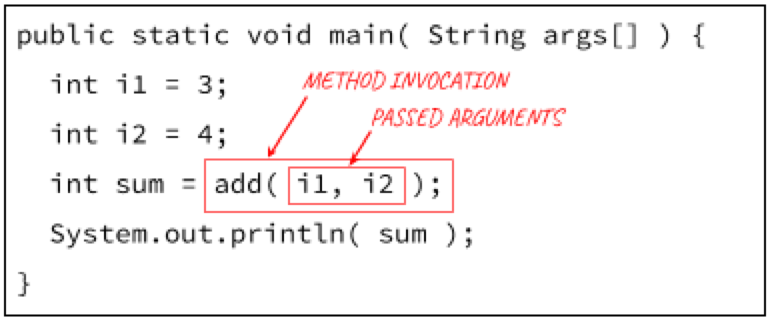


```
public static int add ( int i1, int i2 ) {
    int sum = i1 + i2;
    return sum;
}
```

Fig 0.0 - A method definition for a simple addition operation.

For the purposes of this TIP, we are concerned primarily with the return statement.

Argument Passing:
When a method is invoked (also sometimes referred to as 'calling' or 'using' a method), appropriate arguments must be passed in to the method based upon the method's signature.



```
public static void main( String args[] ) {
  int i1 = 3;
  int i2 = 4;
  int sum = add( i1, i2 );
  System.out.println( sum );
}
```

Fig 0.1 - Arguments being passed to a method during invocation

Argument Returns:
Where a method definition defines a return value, this is the value that the method invocation resolves to upon completion.

```
sum = add( i1, i2 );
sum = ( 7 );
```

Fig 0.2 - Because 'add(`i1, i2`)' returns the value (`7`), these two lines of code have the same effect.

Note:
If a method's return value is not properly assigned at the time of invocation, it will be lost.

Program Counter:
Also referred to as the Instruction Pointer (IP), the Program Counter is a tool used by the JVM to "remember its place" while it executes code instructions. This is akin to how you might place a finger on a page while reading a book.

Fig. 0.3 - The human practice of noting one's place while reading is analogous to the JVM's use of a program counter.

In order to hop from one position in the book to another (such as an index or glossary), you might note your page and line number on a pad of paper before flipping the pages and placing your finger on the next page you're interested in. By using a pad of paper (or dedicated memory space), you can keep track of a very large number of such hops and always find your way back whenever you want.

1 - Problem Statement:

A method includes a call to itself as a part of its definition. What happens? Is this even legal?

```
public static int recurse( int i ) {

  if( i <= 0 ) {
    return i;
  }

  else {
    i--;
    int retVal = recurse( i ); ⇐ SELF-REFERENTIAL METHOD CALL
    return retVal;
  }

}
```

First, yes, this is entirely legal. Methods which call themselves are examples of **recursive** (sometimes called 'inductive') operations. While we tend not to think recursively by default, we're usually able to recognize recursive patterns when they appear in nature.

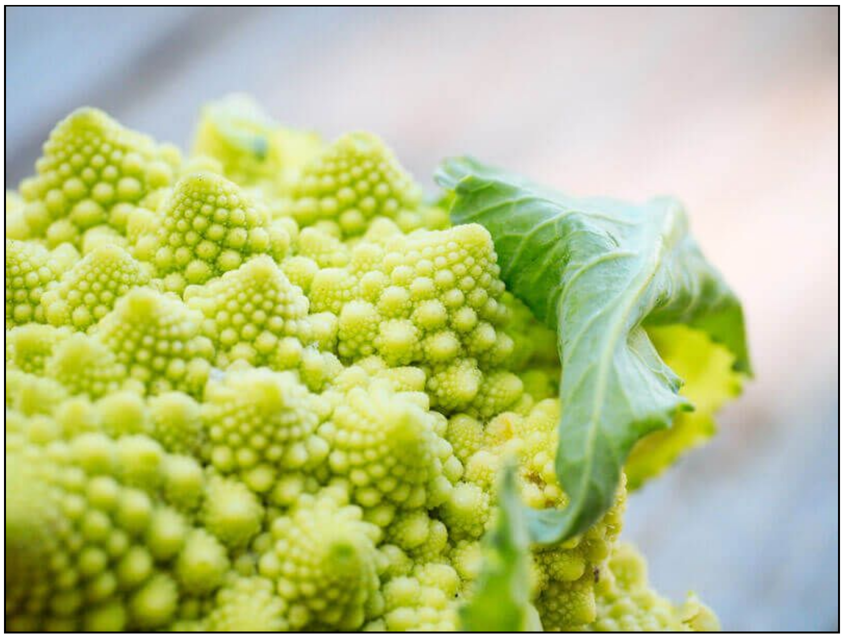
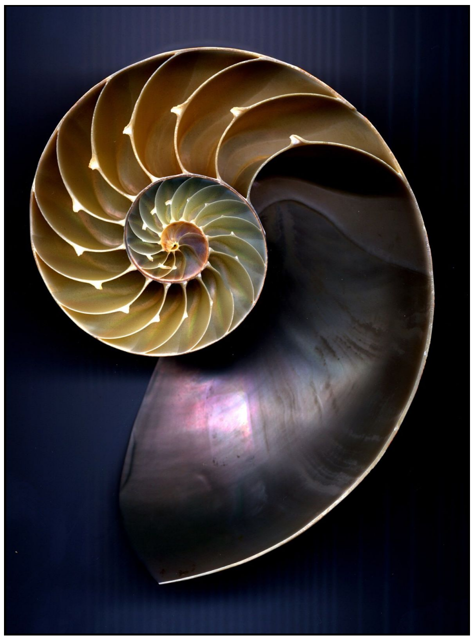
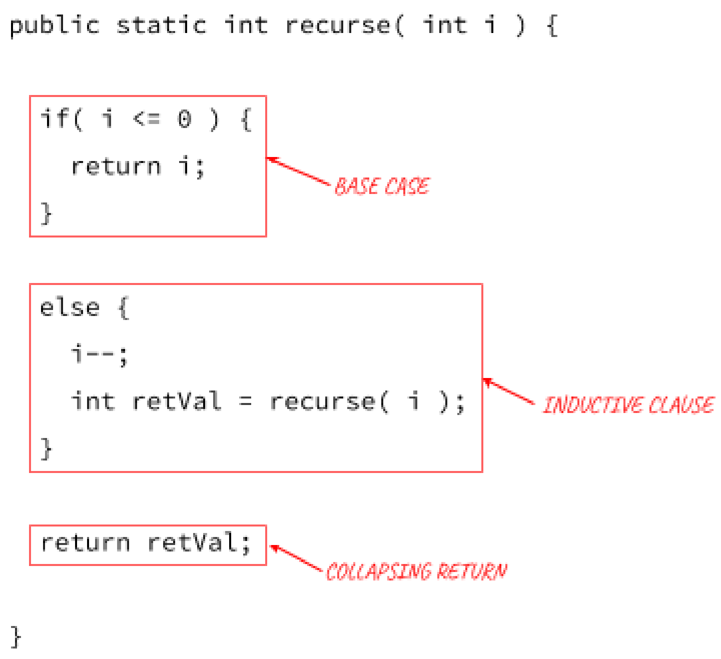



Fig. 1.0 - Romanesco broccoli and nautilus shells are two examples of recursive morphologies wherein the shape of successive growth is related to the prior "layer".

In society, you may be familiar with 'viral' memes, geometric growth, percentage interest, or Ponzi schemes. These are all examples of recursive phenomena.

In computer science, a properly defined recursive method is usually composed of two components:

1) The **Base Case** - Which describes the conditions to terminate the recursion
2) The **Inductive Clause** - The action performed by a recursive method



```
public static int recurse( int i ) {

  if( i <= 0 ) {
    return i;
  }

  else {
    i--;
    int retVal = recurse( i );
  }

  return retVal;

}
```

Fig 1.1 - A simple recursive function with the base case and inductive clause highlighted.

As for the question of "what happens?", this is somewhat more difficult to answer and is an emergent property of the action(s) performed by the inductive clause and the conditions that trigger the base case. Unfortunately, there isn't usually a simple means of determining the end result besides careful thought and/or running the method and examining the results.

Note:
All recursive functions can be implemented iteratively, which is to say that they may be rephrased such that identical behavior is achieved without a self-referential method call. Iterative equivalents may require additional support variables and extra lines of code to implement.

*However, because recursive methods are reliant on the available size of the **call stack**, there may be instances with large bodies of data or a deeply recursive calls where an iterative equivalent is preferred or necessary.*

2 - Applicability:

Given the apparent added complexity of recursive thinking, you may wonder why one would engage with recursion at all. Indeed, many programmers are able to go through their careers without writing recursive functions, and some companies even go so far as to ban the use of recursion in their codebases.

At the same time, for certain problems, recursive methods form the most succinct (and by extension, most maintainable) solutions available. In an academic context, several single-player games are often used to illustrate the process of converting a recursive method to useful work.

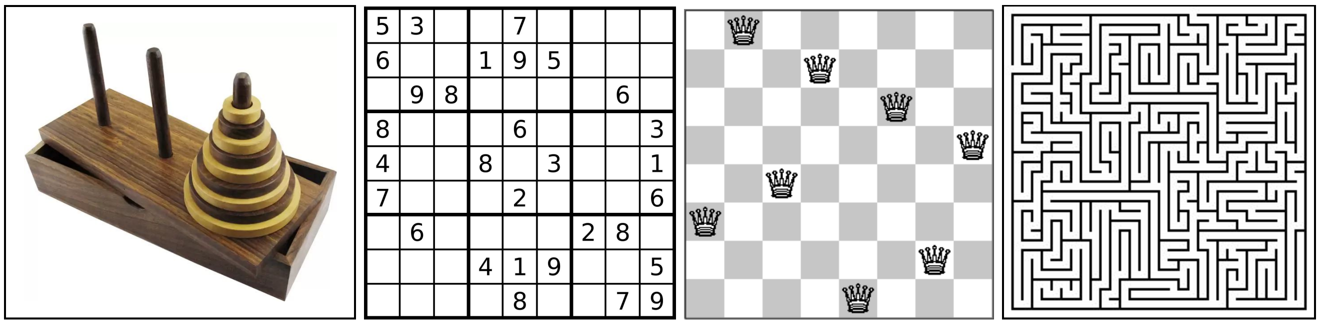


*Fig 2.1 - Several games, including the towers of Hanoi, Sudoku, the 8-Queens problem, and 2-D mazes may be solved using a particular variant of recursion called recursive **backtracking**.*

If using a **recursive backtracking solution** to these games, there are just three rules:
- The base case is a successful "win" state, which terminates the process
- While the inductive clause:
  - Makes the systematic "next" possible move OR
  - Where no further moves are available, backtracks the game state until another move can be explored

Using these three conditions, the entire **state space** of these games can be explored until the base case is reached and the problem is solved. Beyond these games, the same strategy may be applied to an entire class of problems, known as **state space searches**, with applications as wide ranging as strategic planning, traffic coordination, financial planning, and wayfinding (to name a few).

In computer science, you are also likely to encounter recursion during a class of problems called **traversals**. In a traversal problem, you must move across a body of data in a systematic fashion, usually performing work along the way or searching for a value or condition.

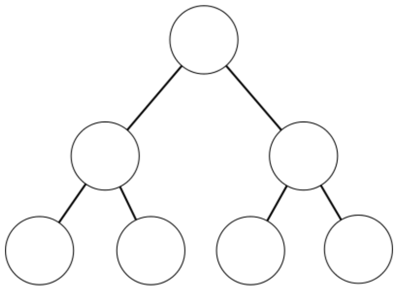
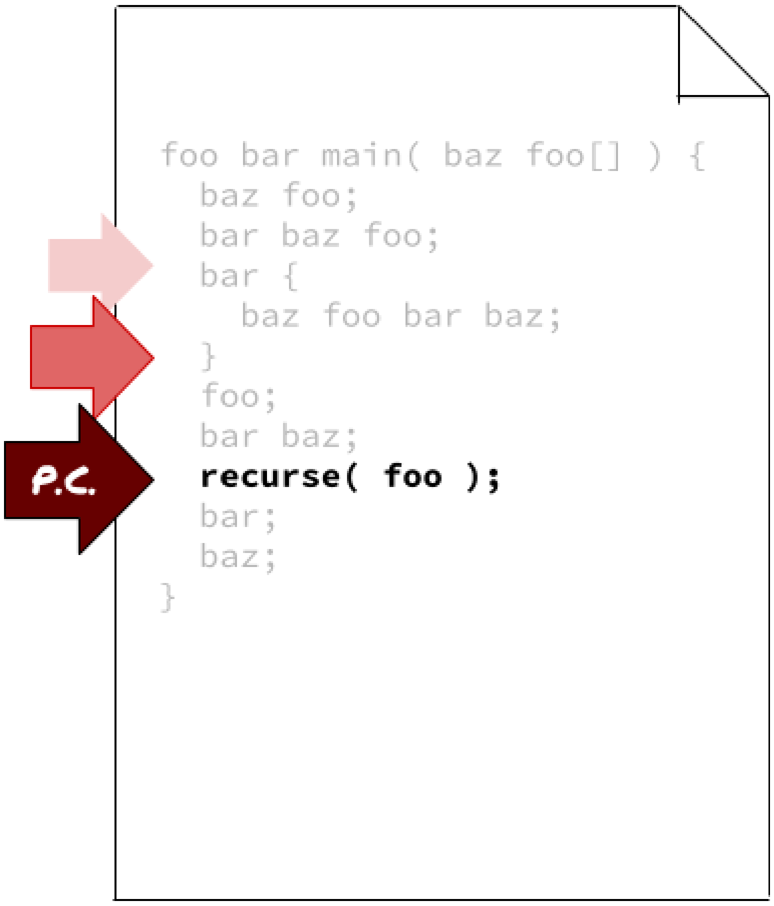


*Fig 2.2 - Binary tree traversals are one of the most common applications of recursion. You'll encounter these when studying the properties and uses of **binary trees**.*

Dynamic problems also tend to lend themselves to recursive solutions. In dynamic problems, it is generally not possible (or at least quite difficult) to calculate a solution directly. Instead, a solution must be found by calculating successive intermediary steps (called 'sub-problems') until the solution is reached. One of the classic examples of dynamic problems is finding the Nth value in the Fibonacci or Tribonacci sequence, though the one you'll probably spend the most time on in school will be **Djikstra's Algorithm** for finding the shortest path through a graph (but don't worry at all about it right this moment). Other applications include fluid dynamics calculations, weather predictions, orbital calculations, and a host of simulation scenarios.

3 - Mental Model:

Before attempting to address the effects of a recursive method, we should first establish a firm understanding of how the Java Virtual Machine (JVM) will handle it, mechanically-speaking. Let us first translate the program logic to a physical medium such as a sheet of paper. On this sheet, your program looks the same as in your programming environment, and you should already know that a Java program just uses a program counter to read down the sheet line by line, executing the instructions as they are encountered.



```
foo bar main( baz foo[] ) {
  baz foo;
  bar baz foo;
  bar {
    baz foo bar baz;
  }
  foo;
  bar baz;
  recurse( foo );
  bar;
  baz;
}
```

Fig 3.0 - General model of how the program counter proceeds through a program up until a method is called.

When a method is encountered, execution down this sheet must halt until the method is resolved. An analogue may be achieved by writing the method definition upon a notecard and then:

(1) - Placing the card atop the original program sheet
(2) - Copy the passed argument values onto the card
(3) - Proceed through the method line by line until the return is encountered
(4) - Pass the return value back to the calling location

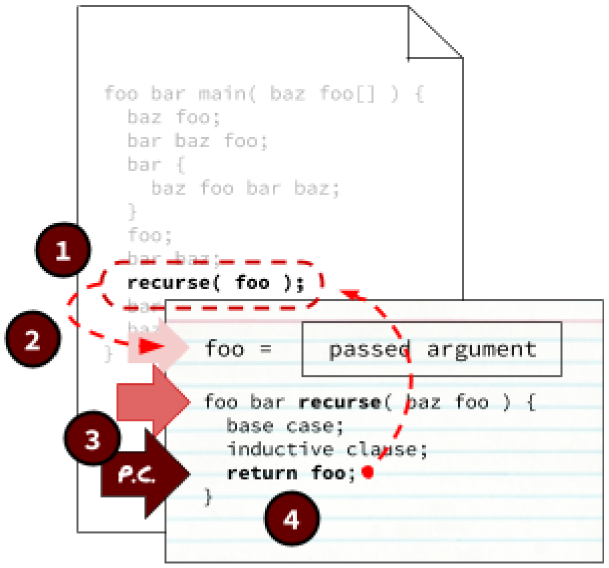


Fig 3.1 - General model of a method call. The original call of 'recurse(foo)' will resolve to whatever value is returned.

But what if a method call is encountered while a method is being processed? Java will use the same strategy, applying another notecard with the other method definition atop the first card, then process it to completion or a return.

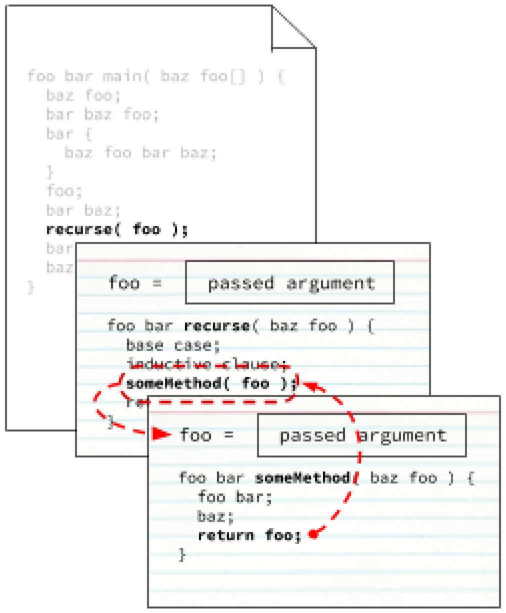
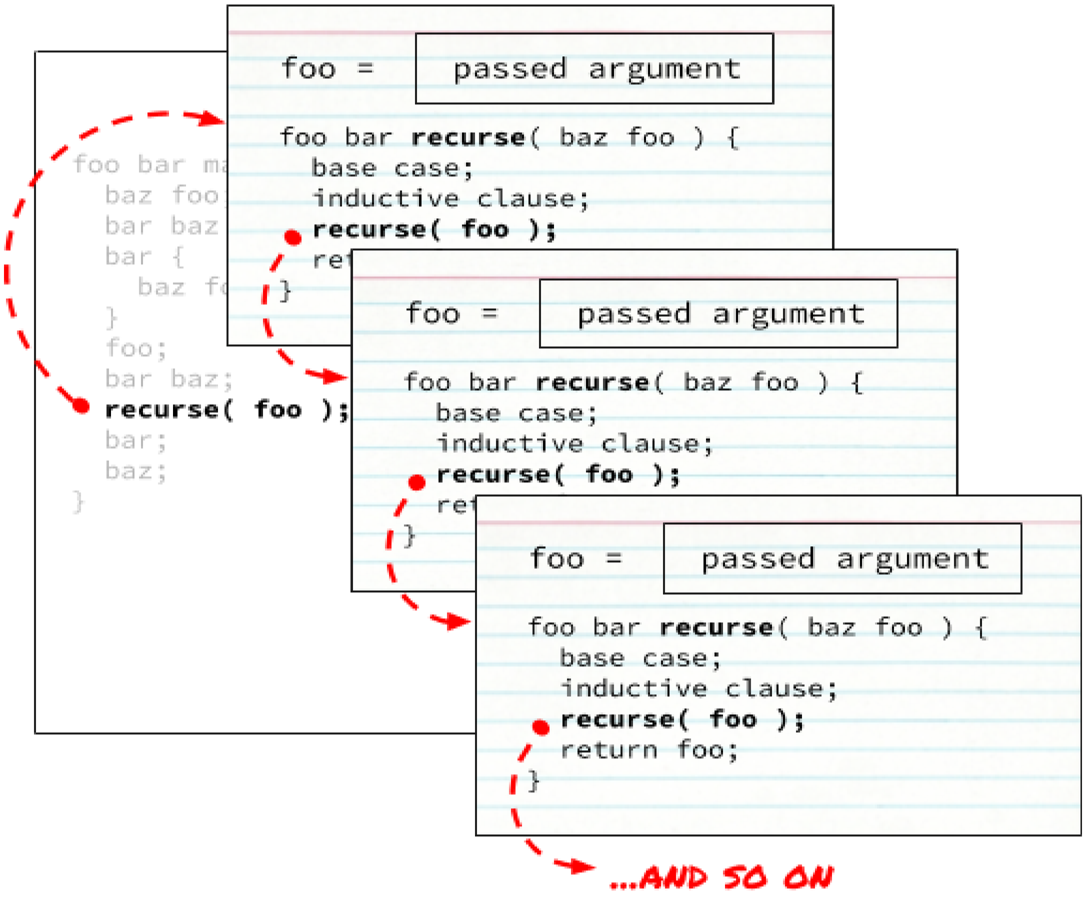


Fig 3.2 - A model for nested method calls. A method card is placed on the stack and processed to completion. If a return exists, the call resolves to the value of the return.

In a recursive function, the **only** difference is that the second method called happens to have the same definition as the first. Besides this detail, they are processed in exactly the same fashion where:

- A notecard containing the method definition is made
- Passed arguments are copied to the new notecard
- The card is processed line by line until it completes/returns or makes another method call
- If it completes on a return, a value is sent back to the card below
- Otherwise, the card is simply removed from the stack



```
foo bar ma
   baz foo
   bar baz
   bar {
      baz fo
   }
   foo;
   bar baz;
   recurse( foo );
   bar;
   baz;
}
```

```
foo =   passed argument

foo bar recurse( baz foo ) {
   base case;
   inductive clause;
   recurse( foo );
   re
}
```

```
foo =   passed argument

foo bar recurse( baz foo ) {
   base case;
   inductive clause;
   recurse( foo );
   re
}
```

```
foo =   passed argument

foo bar recurse( baz foo ) {
   base case;
   inductive clause;
   recurse( foo );
   return foo;
}
```

Fig 3.4 - Generalized model of program flow during recursive calls

At this point, you're probably noticing the unnerving possibility that such a series of calls may proceed forever. Such a process is called '**infinite regress**'.

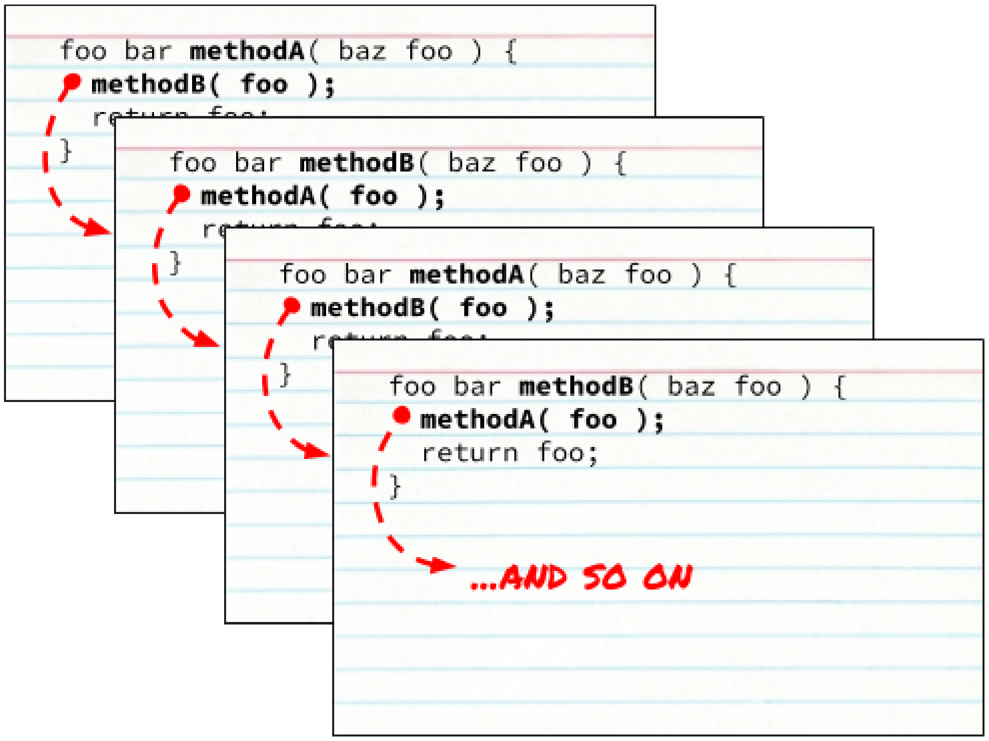
Note:

Infinite regress *describes a series of elements or propositions that have a start and associated next elements, but no last element. It should be distinguished from* ***circular logic****, in which elements eventually lead back upon themselves.*

While both conditions are usually to be avoided in computer science and have similar effects (the program will generally fail to progress), infinite regresses will tend to cause a crash as memory resources are exhausted while circular logic can in theory (and practice) run forever.

A 'while' loop waiting on an event may be an example of a desirable and deliberate implementation of circular logic.

It bears mention, however, that such regress is no worse than two functions 'A' and 'B' that call each other:



```
foo bar methodA( baz foo ) {
  methodB( foo );
}
```

```
foo bar methodB( baz foo ) {
  methodA( foo );
}
```

```
foo bar methodA( baz foo ) {
  methodB( foo );
}
```

```
foo bar methodB( baz foo ) {
  methodA( foo );
  return foo;
}
```

Fig 3.5 - Infinite regress is not unique to recursive functions. It is entirely possible to achieve this in other circumstances.

In order to avoid this situation, the recursive function should possess a means of exiting the otherwise endless cycle of self-referential method calls. The role of terminating trigger is served by the **base case**, referenced in section 1.

The sole purpose of the base case is to react to a specified condition and trigger a '`return`' **before** or **instead of** the recursive method call.

Don't worry about the specifics of how this is achieved just yet; we'll get to that in 'Section 4 - A Specific Case'.

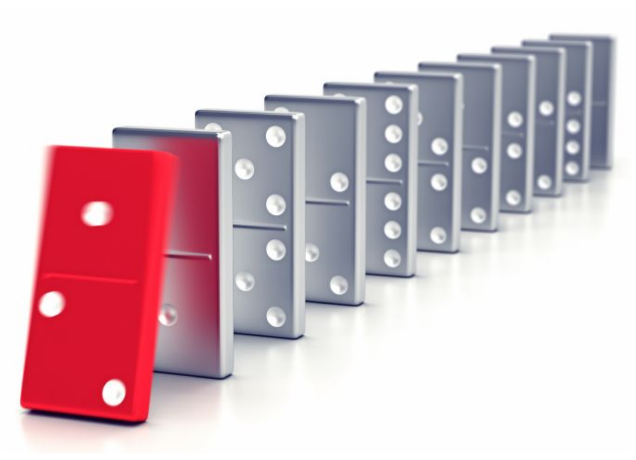
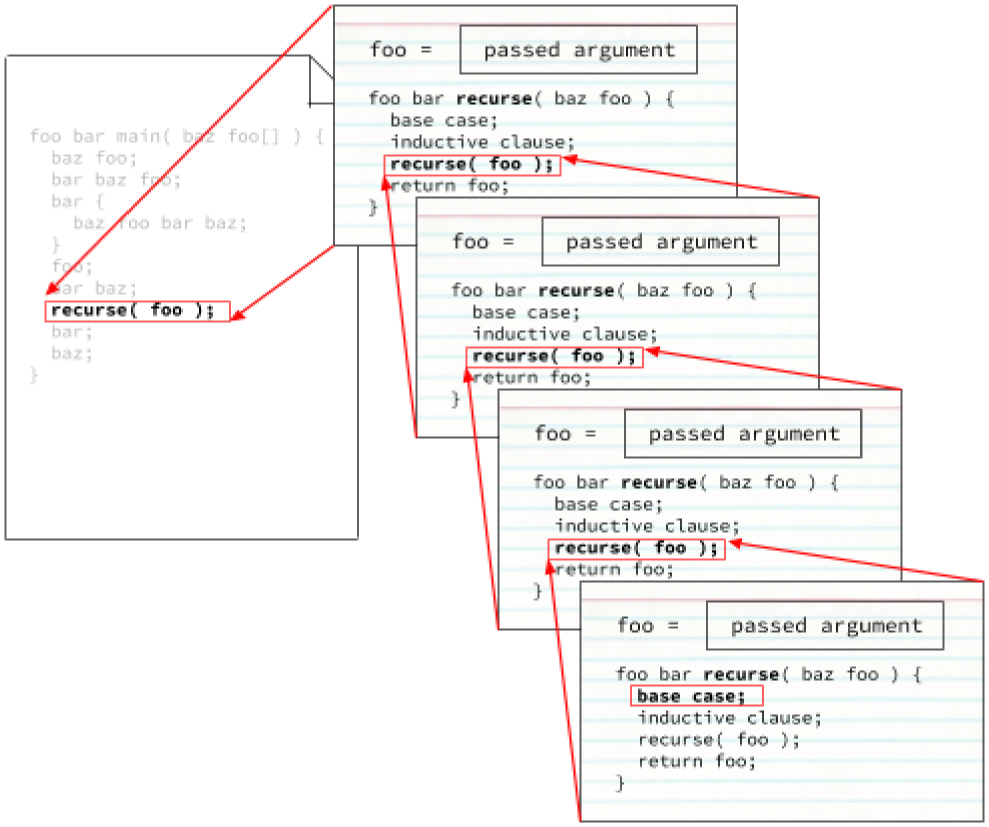


Fig 3.6 - The process of setting up a recursive call stack is, in principle, very similar to the game of dominos.
The base case acts like the final domino in a line, denoting an end to further stacking.

By generating a return value instead of another recursive call, the closing method allows the previous card in the stack to reach its own return value, which in turn allows the next prior card to resolve, and so on until the original method invocation resolves to a value.



```
foo bar main( baz foo[] ) {
  baz foo;
  bar baz foo;
  bar {
    baz foo bar baz;
  }
  foo;
  bar baz;
  recurse( foo );
  bar;
  baz;
}
```

```
foo =  passed argument

foo bar recurse( baz foo ) {
  base case;
  inductive clause;
  recurse( foo );
  return foo;
}
```

```
foo =  passed argument

foo bar recurse( baz foo ) {
  base case;
  inductive clause;
  recurse( foo );
  return foo;
}
```

```
foo =  passed argument

foo bar recurse( baz foo ) {
  base case;
  inductive clause;
  recurse( foo );
  return foo;
}
```

```
foo =  passed argument

foo bar recurse( baz foo ) {
  base case;
  inductive clause;
  recurse( foo );
  return foo;
}
```

Fig 3.7 - Collapse of the recursive stack triggered by the base case activating.

Fig 3.8 - Successive completions and closures of method calls cause the call stack to collapse all the way back to its start. If the method possesses a return value, information from the base case can ride this collapsing wave to the top of the stack.

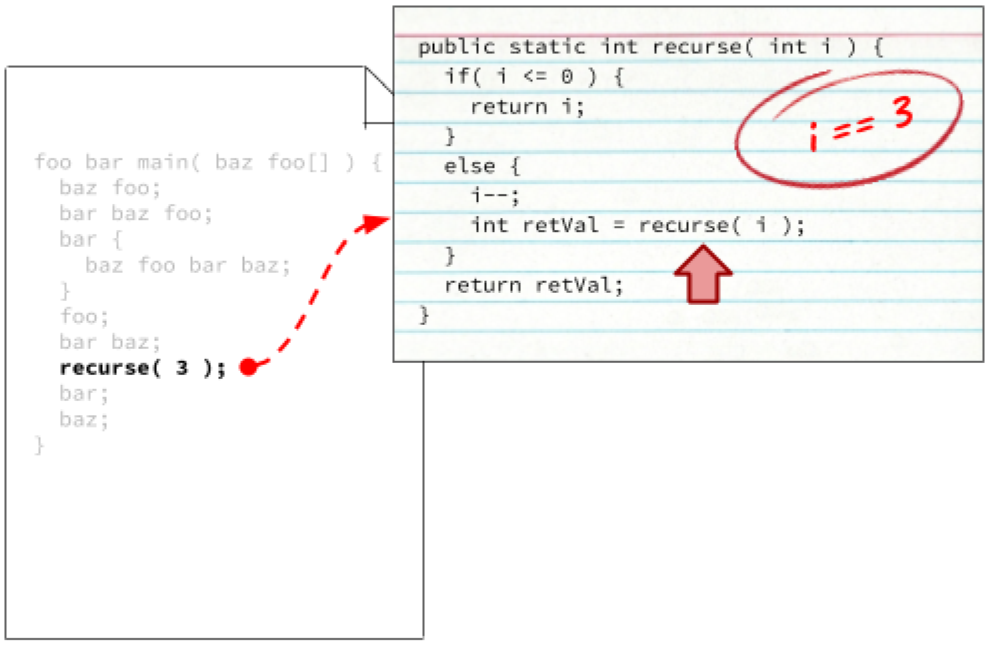
4 - A Specific Example:

With the above mental model in mind, let's turn back to the original question of what the function '`recurse()`' might actually do. Recall that the method from 'Section 1 - Problem Statement' was defined as follows:

```
public static int recurse( int i ) {

  if( i <= 0 ) { ⇐ BASE CASE
    return i; ⇐ BASE RETURN
  }

  else { ⇐ INDUCTIVE CLAUSE
    i--;
    int retVal = recurse( i );
    return retVal; ⇐ RELAY RETURN
  }

}
```

The simplest means of investigating this function's operation is to just attempt it - by hand - using an arbitrary value of '`i`'. Here, let's try `(3)` and see what happens. On the first call to recurse:

- Java enters the method with '`i`' set to `(3)`
- The base case does not trigger (3 is not less than or equal to 0)
- Moving to the inductive clause (the `else` statement), '`i`' is decremented by 1 (so it is now `(2)`)
- A return variable is declared and assigned the yet-to-be resolved value of '`recurse( 2 )`'



```
foo bar main( baz foo[] ) {
  baz foo;
  bar baz foo;
  bar {
    baz foo bar baz;
  }
  foo;
  bar baz;
  recurse( 3 );
  bar;
  baz;
}
```

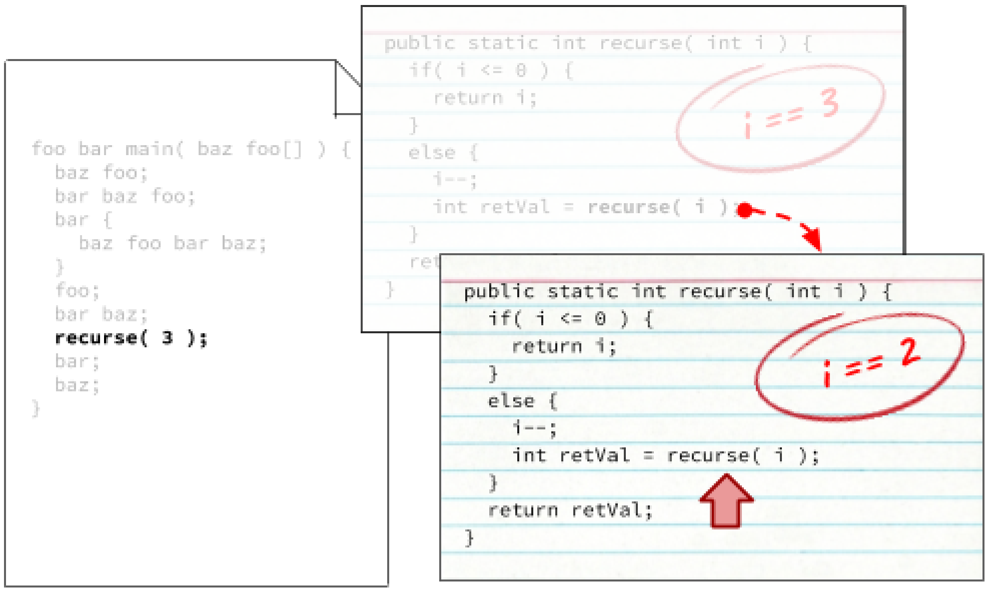
```
public static int recurse( int i ) {
  if( i <= 0 ) {
    return i;
  }
  else {
    i--;
    int retVal = recurse( i );
  }
  return retVal;
}
```

Fig 4.1 - The first call to '`recurse( 3 )`' is made and processed until the recursive method call

Here, the method must pause since it does not yet know what '`recurse( 2 )`' resolves to.

In order to find out the result of 'recurse(2)', it generates a second method call atop the first which behaves in a very similar fashion to the previous call:

- Java enters the method with 'i' set to (2)
- The base case does not trigger (2 is not less than or equal to 0)
- Moving to the inductive clause (the else statement), 'i' is decremented by 1 (so it is now (1))
- A return variable is declared and assigned the yet-to-be resolved value of 'recurse(1)'

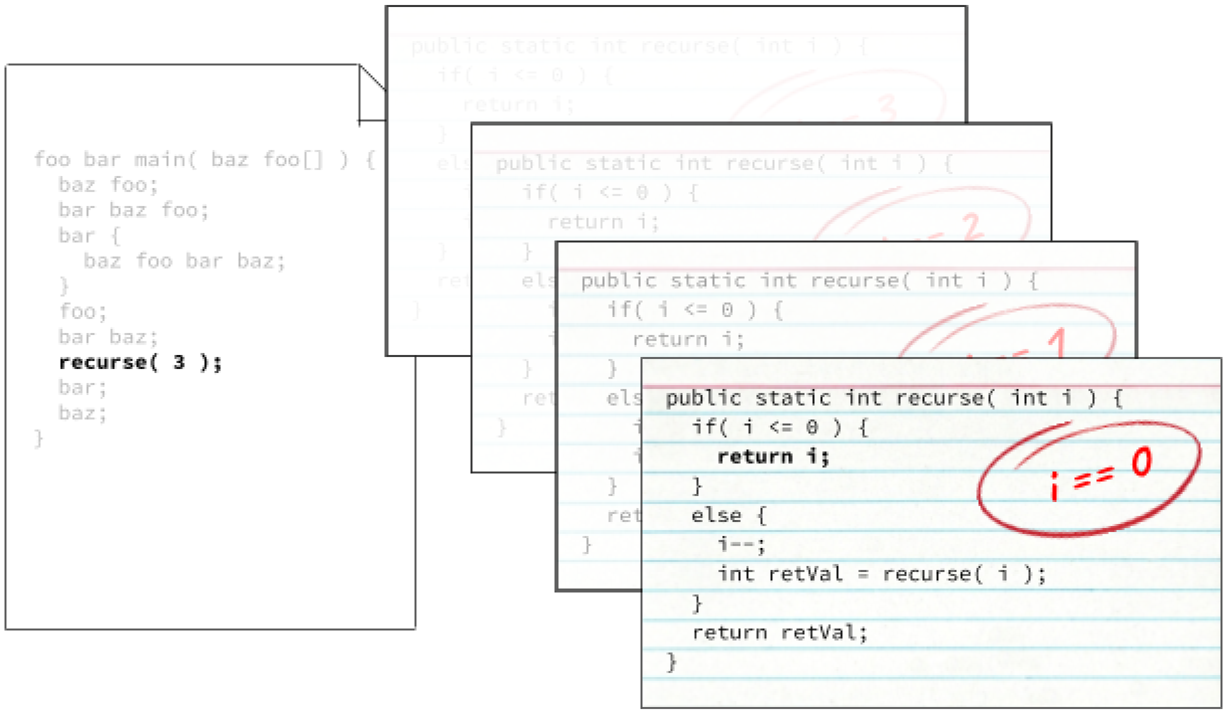


```
foo bar main( baz foo[] ) {
  baz foo;
  bar baz foo;
  bar {
    baz foo bar baz;
  }
  foo;
  bar baz;
  recurse( 3 );
  bar;
  baz;
}
```

```
public static int recurse( int i ) {
  if( i <= 0 ) {
    return i;
  }
  else {
    i--;
    int retVal = recurse( i );
  }
  return retVal;
}
```

Fig 4.2 - The second call to 'recurse(2)' is made and processed until the recursive method call

This process continues until a call to recurse() is made with an argument of (0).



```
public static int recurse( int i ) {
  if( i <= 0 ) {
    return i;
  }
  else {
    i--;
    int retVal = recurse( i );
  }
  return retVal;
}
```

Fig 4.3 - Recursive calls are made until a call to 'recurse(0)' occurs

At this point, something special happens! As we trace the execution, you can see that the base case is triggered ('`i`' - being `(0)` - is indeed less than or equal to 0). This allows this call to return instead of executing another call to '`recurse()`'.

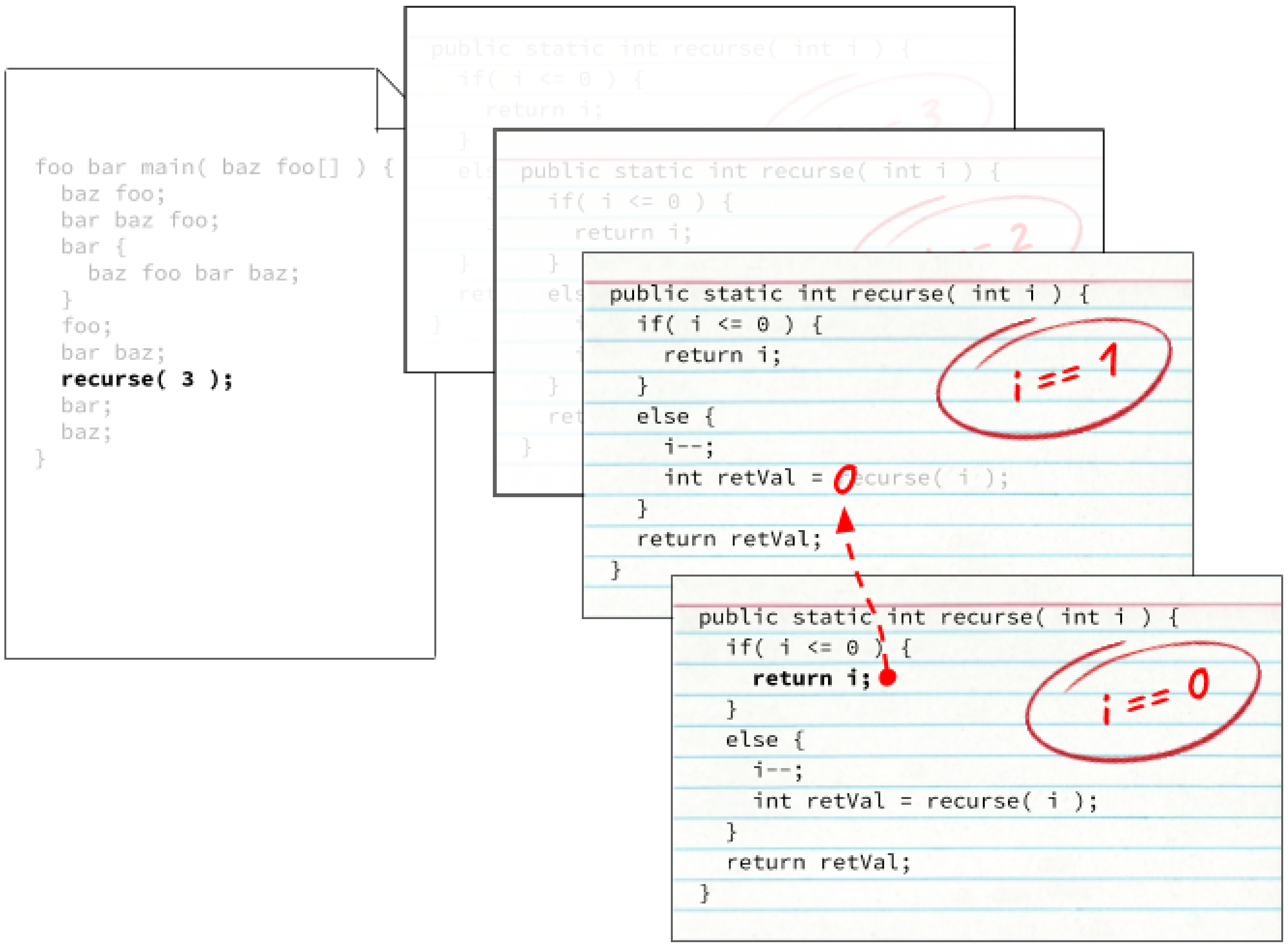
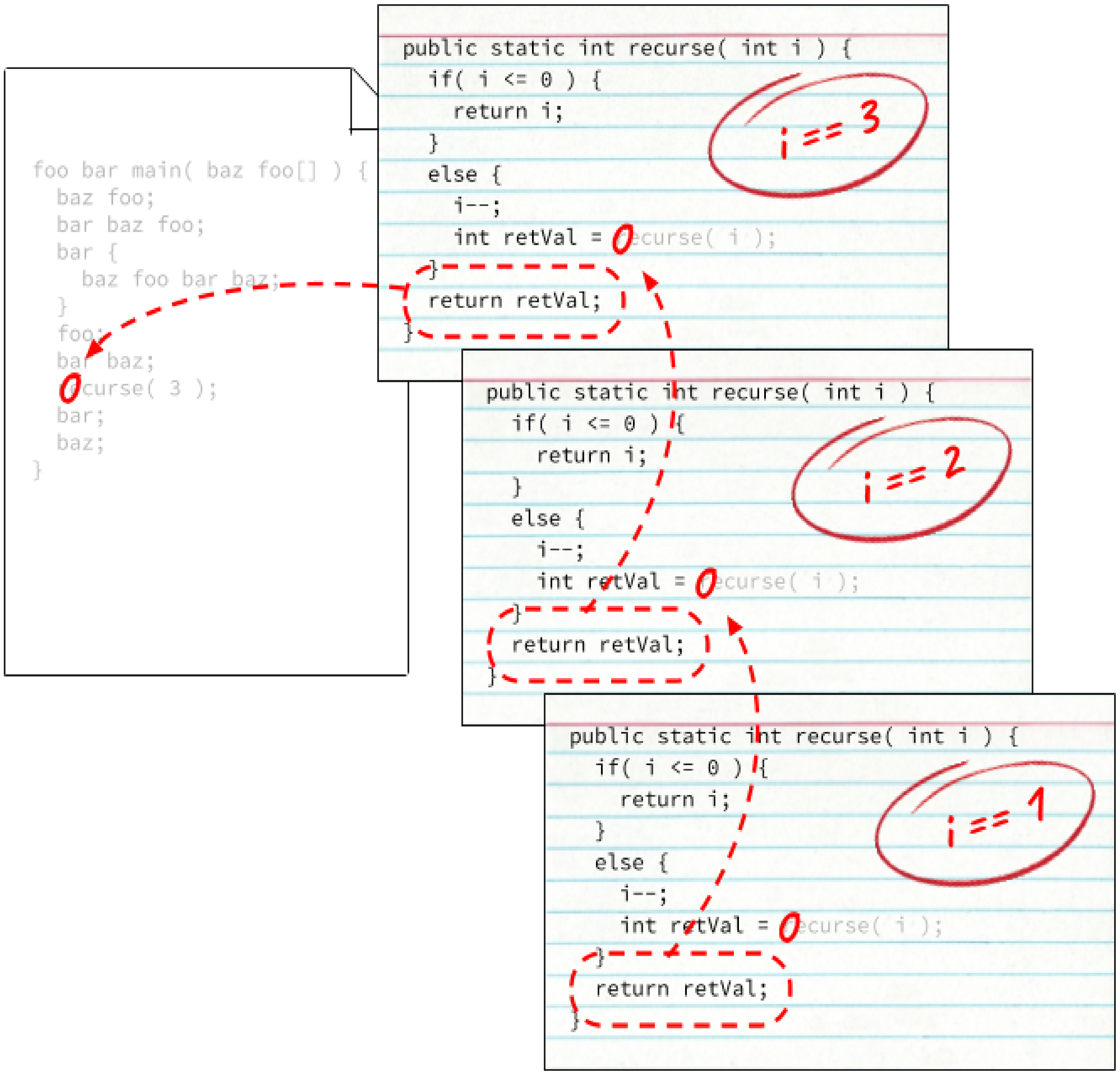


Fig 4.4 - The base case generates the first call closure in this process and passes a return value to the prior method in the stack.

The prior call to '`recurse( 0 )`' now resolves to `(0)`, allowing the previous method call to run to completion.

In turn, a successful return allows the next call to resolve, and so on and so forth until the original invocation of '`recurse( 3 )`' receives an answer.



```
foo bar main( baz foo[] ) {
  baz foo;
  bar baz foo;
  bar {
    baz foo bar baz;
  }
  foo;
  bar baz;
  recurse( 3 );
  bar;
  baz;
}
```

```
public static int recurse( int i ) {
  if( i <= 0 ) {
    return i;
  }
  else {
    i--;
    int retVal = recurse( i );
  }
  return retVal;
}
```

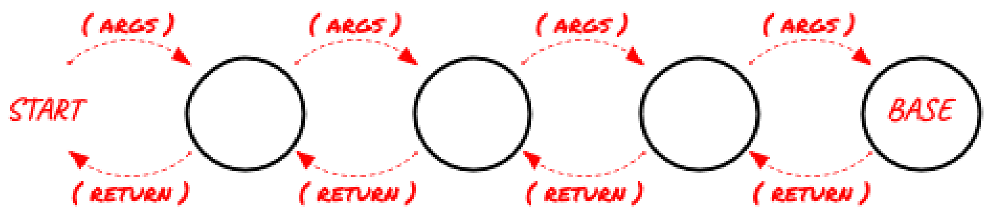
Congratulations! You've just determined that a call to '`recurse()`' will tend to reduce a number to 0!

Question:
What effect does a call of '`recurse( -5 )`' yield?
How many layers of recursion are required to arrive at an answer?

5 - Representation:

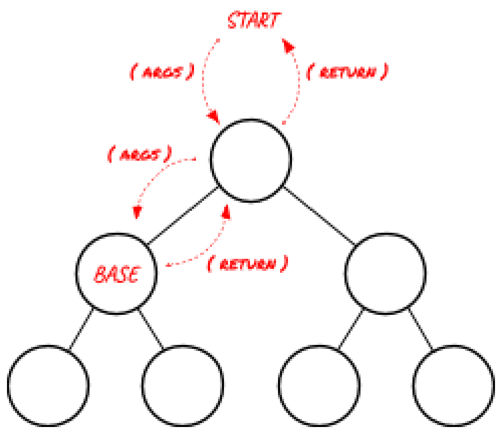
While accurate, time constraints (as on an exam) may prevent you from modeling a series of recursive calls in the above fashion using notecards. Multiple shorthand approaches and conventions exist, and you should endeavor to use and become comfortable with a reasonable notation for describing recursive call stacks and their effects.

One possible example could be the following:

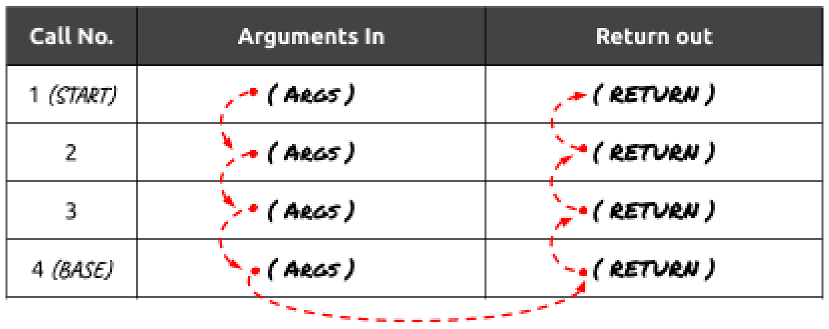


Here, each large circle (node) represents a recursive method call and the arguments passed in are represented in a coordinate format. Returns are similarly represented.

In the case of tree traversals or tree operations, the tree itself may be used to provide this structure:



As you grow more comfortable with visualizing recursion, it's even possible to tabulate the recursive calls into a table format:



Call No.	Arguments In	Return out
1 (START)	(ARGS)	(RETURN)
2	(ARGS)	(RETURN)
3	(ARGS)	(RETURN)
4 (BASE)	(ARGS)	(RETURN)

However, there is no established industry standard. Find a method that works for you, but keep in mind that this process may be represented in a variety of formats.

6 - Recursive Patterns:

While careful thought (and running the code) remains the most effective means of analyzing recursive methods, there are certain popular recursive patterns you may start to notice including:

Returning the base case - A value at or near the base case is isolated and returned up the call stack
Counting down - A counter variable is sent down to the base case; its value is returned back up
Counting up - The base case establishes a counter which is modified as it returns
Pre-order data export - Data is exported from the method (as to the console) **before** the recursive call
Post-order data export - Data is exported from the method (as to the console) **after** the recursive call
State set/search - A data state is systematically changed with no return; recursion may halt on a 'win' condition
Divide and Conquer - A large problem space is reduced geometrically until a single, solvable problem remains

This list is by no means exhaustive, but getting a broad sense of what the code author was trying to do can be a good first step in recursion analysis. Three helpful questions to ask are:

1) The base case - Under what circumstances does recursion halt? What output does the base generate?
2) The inductive clause - What action(s) does the method perform before and/or after the recursive call?
3) The return - What does the method return? Is the return modified as it makes its way back to the start?

Note:
Be aware that the complexity of methods - including recursive ones - can be arbitrarily high. The base case may not be clearly defined (or there may be multiple base cases), the inductive clause may include logical branches, the clauses may be placed in any order, and even multiple recursive functions may be intertwined together.

In cases like these, just remain calm and know that you can always process the calls one line at a time.

7 - Practice Problems:

1) Write a recursive function that increments or decrements an integer endlessly. The function should eventually result in an error or program crash.

Hint:
You may highlight the methods to reveal a possible solution. Note that many solutions exist.

```
public static void recurse( int i ) {


}
```

2) Add a base case to problem 1's solution to halt recursion when `i == 0`.

```
public static void recurse( int i ) {
   if(          ) { // BASE CASE

   }
   else { // INDUCTIVE CLAUSE


   }
}
```

Note:
*Even though logically an incrementing recursive stack will eventually reach the value (0) (either directly or by **integer overflow**), a crash may still occur if a limit is set on the number of **nested calls** allowed in the running environment. In Java, you may expect the JVM to allow several thousand nested method calls before throwing an error.*

3) Add a second variable (a counter) in the recursive method signature to control the base case. Halt recursion when '`counter == 0`'.

```
public static void recurse( int counter, int i ) {
   if(                 ) { // BASE CASE

   }
   else { // INDUCTIVE CLAUSE



   }
}
```

4) Modifying the method from problem 3, return the value of '`i`' at the base case back up through the call stack. Print the value to the console directly prior to any return.

```
public static int recurse( int counter, int i ) {
   if(                 ) { // BASE CASE


   }
   else { // INDUCTIVE CLAUSE





   }
}
```

5) Using what you've learned in problems 1 to 4, write a recursive method that increments an int '`i`' exactly '`counter`' times. The call:

```
int i = recurse( 10, 20 );
System.out.println( );
System.out.println( "Result: " + i );
```

Should generate a similar output to:

```
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
Result: 30
```

```
public static int recurse( int counter, int i ) {
  if(              ) { // BASE CASE


  }
  else { // INDUCTIVE CLAUSE





  }
}
```

8 - Challenge:

1) Write a method '`decToBin`' that converts an integer from range (`0`) to (`255`) to its equivalent binary representation assuming the following bit sequences:

```
  0 == 00000000
  1 == 00000001
  2 == 00000001
      [...]
253 == 11111101
254 == 11111110
255 == 11111111
```

Your method need not be recursive or generate a return, but it must output the characters to console. The call '`decToBin( 85 )`' should generate a console output of '`01010101`' or '`1010101`' .

```
public static void decToBin( int i ) {








}
```

Note:
Multiple decimal to binary converters are available on the internet. Try using one to validate your outputs.

2) Address the following:
- In your own words, describe the effect of the following method
- Identify the location of the base case and inductive clause
- What is the purpose of the '`bits`' variable in this case?

```
public static void recurse( int bits, int i ) {
  if( bits == 0 ) {
    return;
  }
  else {
    bits--;
    recurse( bits, i/2 );
    System.out.print( i%2 );
  }
}
```

Note:

*In mathematics, the **Most Significant Digit (MSD)** is located to the left of a number while the **Least Significant Digit (LSD)** is located to the right. When writing bit sequences, computer scientists also must think about the **Most Significant Bit (MSB)** and **Least Significant Bit (LSB)**. Since memory is somewhat dyslexic, no consensus has ever been reached on whether the MSB and LSB should be to the left and right, respectively, or vice versa.*

*To achieve specificity regarding this orientation, computer scientists must state the **Endian Order** or **Endianness** of their sequences. There can be **Big Endian** sequences in which the MSB comes "first", or **Little Endian** sequences that are the opposite with the LSB coming "first".*

*As if this wasn't challenging enough, the **Byte Endianess** may differ from the **Bit Endianess**, meaning a 4-byte integer may need to be interpreted from left-to-right while each byte's bits interpreted from right-to-left!*

3) Address the following:

- What is the effect of the following method?
- Is the method recursive?
- (Bonus) Describe how this effect is achieved.

```
public static String method( int i ) {
  return ( i == 0 ) ? ( "0" ) : (( i/2 != 0 ) ? ( method( i/2 ) + i%2 ) : ( ""+i ));
}
```

Note:

*A **Ternary Operator** in Java is similar to an if-else clause, but takes a different format. It is defined by the use of a '?', followed by two clauses.*

```
<Logical Test> ? <True Clause> : <False Clause>
```

If the Logical Test evaluates to 'True', then the logic in the True Clause will activate. Otherwise, the False Clause will run.

Be aware that ternary operators, like if-else-then clauses, can be nested.

4) Compare the methods of your answer in Question 1 to the method in Question 3. Summarize the pros and cons that you can think of to each approach.

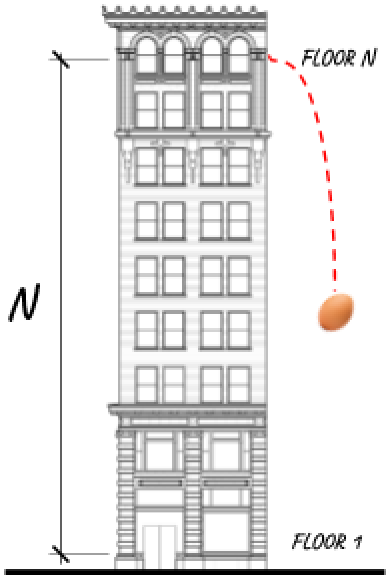
Pros:

Cons:

EXERCISE 01

The 2-Egg problem:
Given two eggs and a building of 'N' floors, you are attempting to determine the strength of the eggs by dropping them from the building. These eggs obey the following rules:

- They possess a strength 'S' from '0' to 'N' which permits them to survive a drop from 'N' floors up
- Both eggs have the same strength 'S'
- If they are dropped from above floor 'S', they will break and be unusable
- If they are dropped from floor 'S' or below, they are unaffected and may be dropped again



What are the fewest number of drops you can make in order to validate the strength of the eggs with a building of 'N' floors?

Note:
*Assuming you had an infinite number of eggs (or at least log(N) eggs), the ideal search strategy would be a **binary search**. You would proceed by dropping an egg at the middle of your search space, then removing the floors above or below appropriately in response to whether the egg broke or not.*

Alas, this is not a binary search, though similarities exist in the recursive structure of both problems.

Step 1:
Reduce the problem such that you only have **one egg**. What are the fewest number of drops needed to validate the strength of the egg?

Populate the following table:

N (Floors)	Drop Sequence	Min. Number of Drops	Max Strength Verifiable
1	1	1	1
2	1, 2	2	2
3			
4			
5			

Step 2:
Returning to **2 eggs**:
A) Describe your search strategy if the first egg breaks. Where is floor 'S' relative to the drop height?
B) If the first egg breaks, where is the worst case floor 'S'?
C) Describe your strategy if the first egg does not break. Where is floor 'S' relative to the drop height?
D) If the first egg never breaks, where is the worse case floor 'S'?
E) Describe in general terms how you might define the ideal first drop point.

Step 3:
Determine the fewest possible drops needed to fully validate the egg strengths for a building of height 1, 2, and 3, 4, and 5. From where and in what order do you drop the eggs in each case?

N (Floors)	Drop Sequence	Min. Number of Drops
1	1	1
2	1, 2	2
3		
4		
5		

Compare notes with your classmates to ensure that you have the fewest possible drops and the correct sequence.

Step 4:
Rephrase the question: Given 2 eggs, what is the maximum number of floors you can validate with 1, 2, 3, … , D drops? Complete the table:

D (Drops)	DROP SEQUENCE		Max. Number of Floors Verifiable
	Worst Case (Immediate Break)	Best Case (No Breaks)	
1	1	1	1
2	2, 1	2, 3	3
3	3, 1, 2	3, 5, 6	6
4	4, 1, 2, 3		
5	5, 1, 2, 3, 4		
6	6, 1, 2, 3, 4, 5		
7	7, 1, 2, 3, 4, 5, 6		
8	8, 1, 2, 3, 4, 5, 6, 7		
9	9, 1, 2, 3, 4, 5, 6, 7, 8		
10	10, 1, 2, 3, 4, 5, 6, 7, 8, 9		
11	11, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10		
12	12, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11		
13	13, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12		
14	14, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13		

Hint:
Is there a generalized rule to how the worst case changes based on D, the number of drops you have available?
What about for the best case?

What is the minimum number of drops required to determine the egg strength if the building has 100 floors?

Step 5:
Implement a recursive method that returns the minimum number of drops needed to find the strength of the eggs for a building of N floors.

```
static int eggDrop( int dropsThusFar, int floorsLeftToValidate ) {

  // BASE CASE
  if(                            ) {

  }

  // INDUCTIVE CLAUSE
  else {


  }

}

public static void main( String args[] ) { // DRIVER
  int n = 100; // BUILDING HEIGHT
  int dropsNeeded = eggDrop( 0, n );
  System.out.println( "Drops needed for (" + n + ") floors: " + dropsNeeded);
}
```

Hint:
A relatively minimal solution is as long as the space provided above and can be rendered in three lines of code plus the conditional statement of the base case.

Challenge:
Generalize the code solution to allow for a variable number of eggs 'E'.