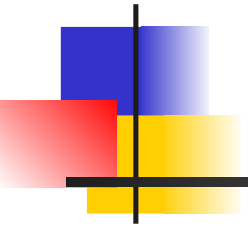


# Recursion

James Brucker

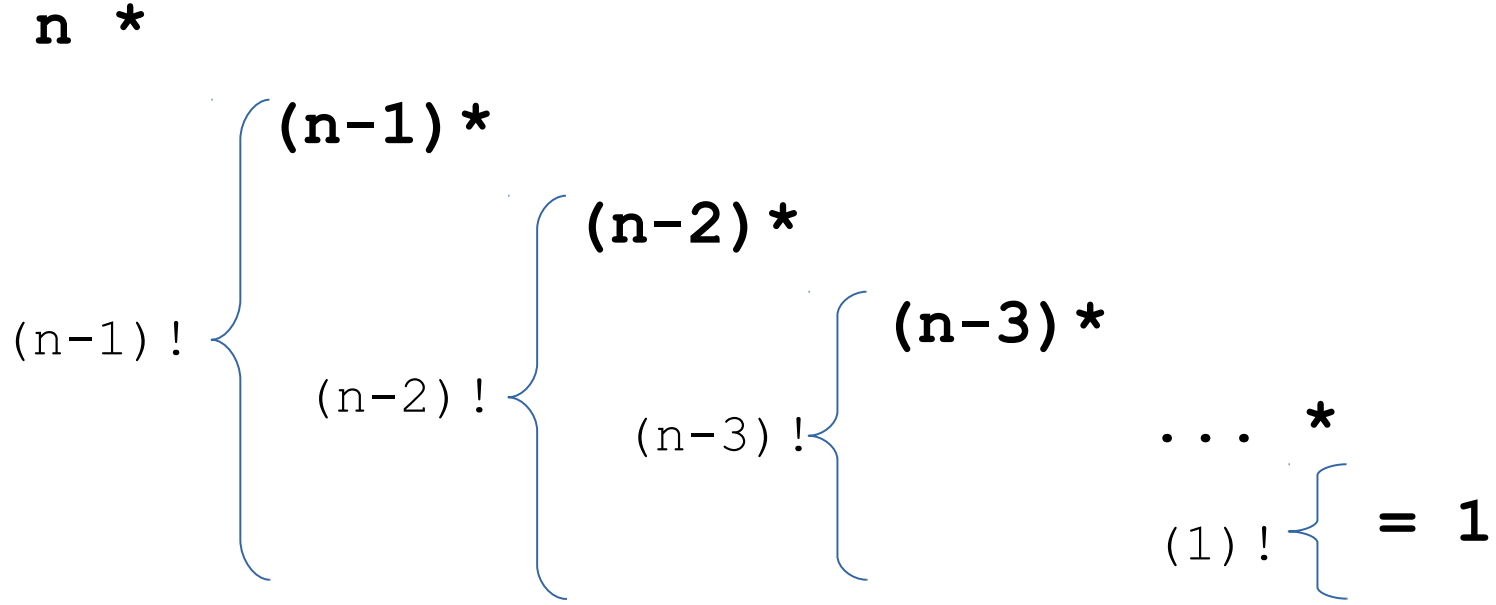
# What is Recursion?

Recursion means for a function or method to call itself.

A typical example of this is computing factorials:

```
n! = n * (n-1)!
```

Using recursion, we can compute n! like this:



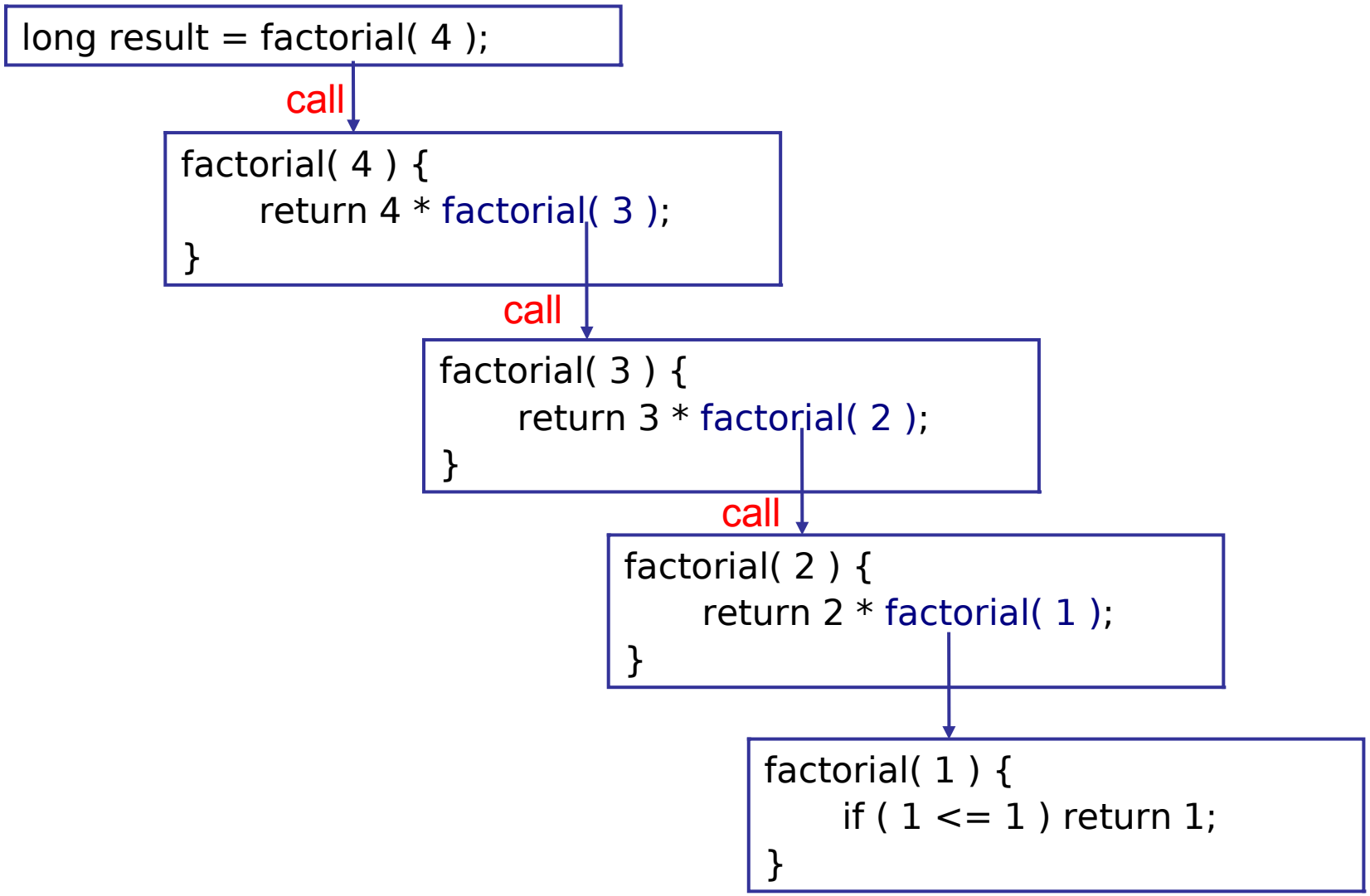
# Recursive factorial(n)

We can write a function that computes factorials by calling itself to compute factorial of a smaller number:

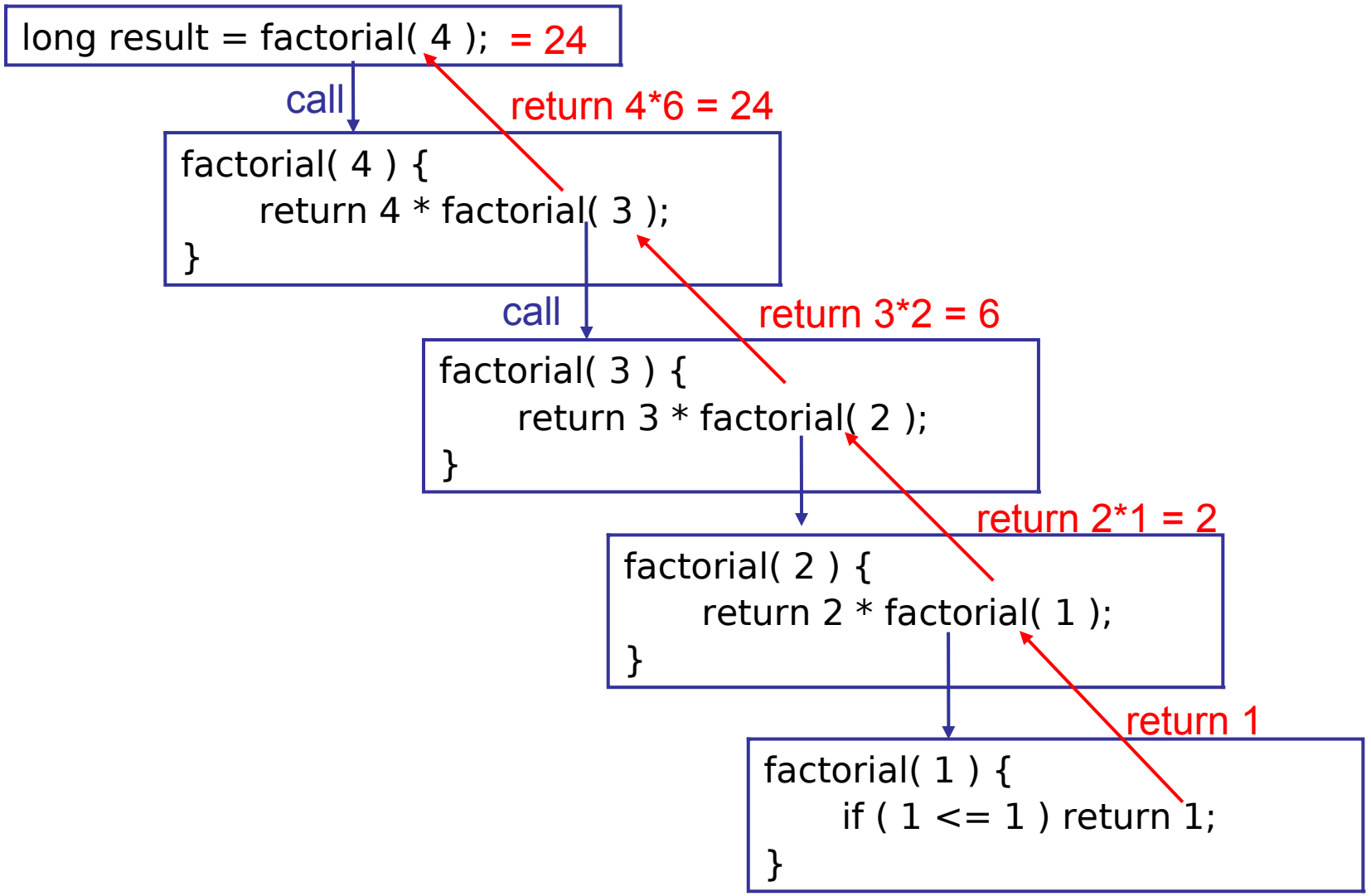
```
long factorial( int n ) {
   if ( n <= 1 ) return 1;
   return n * factorial(n-1);
}
```

Suppose we call this function to compute factorial(4). What statements will be executed?

# factorial(n) execution trace



# factorial(n) return trace



# Important Feature of Recursion

Recursion must guarantee to stop eventually (no infinite calls)

Recursion should not change any state variable that other levels of recursion will use, except by design.

```
long factorial( long n ) {
    if ( n <= 1 ) return 1;
    return n * factorial(n-1);
}
```

This test (n <= 1) guarantees that factorial( ) will eventual stop using recursion.

**Wrong:**

```
long factorial( long n ) {
    if ( n == 1 ) return 1;
    return n * factorial(n-1);
}
```

What happens if factorial(0) is called?

# Base Case

The case where recursion stops is called the **base case**.

factorial(n): base case is n == 1

but you should also test for n < 0

# Recursive Sum

**`long sum(int n)`** - compute sum of 1 to n

Recursion: n + { sum of 1 to n-1 }

Base Case: sum(1) or sum(0)

# Code for recursive sum

Complete this code

```
/**
 * Sum integers 1 to n.
 * @param n largest number to sum, must be
positive.
 * @return the sum
 */
static long sum( int n ) {
   // base case
   if (n <= 0) return 0;
   // recursive case
   return ______?_______; //what should go here?
}
```

# Designing Recursion

1) Discover a **pattern** for recursion:
   - solve a small problem by hand
   - observe how you break down the problem
2) Recursion should provide **insight** and **simplify** the problem.
   - Example: recursive sum does *not* provide insight. Easier and more efficient to use a loop.
3) Determine the **base case** when recursion stops.
4) Termination: What can you **test** to **guarantee** recursion will stop?

# Designing Recursion Example

**`sum(n) = 1 + 2 + 3 + ... + n`**

1) Discover a pattern for recursion:
   - sum(n) = (1 + 2 + ...+ n-1) + n = sum(n-1) + n
2) Does recursion provide **insight** and **simplify?**
   - No -- a loop is easier to understand.
3) **base case**: sum(n) = 0 for any n <= 0.
   Note: to *guarantee* recursion will always stop we need to consider case n < 0, too! Not just n == 0.
   If n < 0 either throw exception or return 0.
4) Guarantee Termination? Yes - each time we call sum(n-1) so parameter value is decreasing sum(3) -> sum(2) -> ...
   Parameter value (n) must eventually be <= 0.

# Recursion using Helper Function

- For some problems recursion is simpler if we define a special function for the recursive call.
  This is sometimes called a "*helper function*".

Example:  sum elements of an array

```
double sum( double [] a ) {

      int n = a.length - 1;
      return a[n] + (sum of a[0] ... a[n-1]);

}
```

# Array sum Helper Function

```
/** sum double[] array using recursion. */
double sum( double a[] ) {
    // use HELPER FUNCTION to sum part of array
    return sumTo( a, a.length-1 );
}
```

sumTo( ) Helper Function sums part of the array.

```
/** Sum elements a[0] + ... + a[lastIndex] */
double sumTo( double a[], int lastIndex )
{
    // base case
    if (lastIndex < 0) return 0.0;
    // recursive case
    return a[lastIndex] + sumTo(a,lastIndex-1);
}
```

# A different base case

What do you think of this helper function?

```
/** Sum elements a[0] + ... + a[lastIndex] */
double sumTo( double a[], int lastIndex )
{
    // base case
    if (lastIndex == 0) return a[0];
    // recursive case
    return a[lastIndex] + sumTo(a,lastIndex-1);
}
```

*Can you think of any case where this may fail?*

# Learn more about Helper Functions

*Big Java*, Chapter 13 (*Recursion*) has a section on helper methods.

# Recursion uses more memory

- We can easily sum 1 to 1,000,000,000 using a loop. but recursion will fail with "out of memory" error.

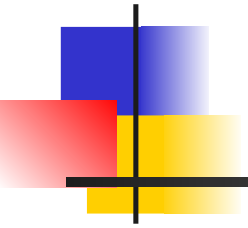
- Why?
  - each function call creates a ***stack frame*** to store information about the invocation (parameters, local vars, saved registers) and return value.
  - The stack frames consume memory on the "stack".
  - Eventually, recursive calls may fill all the stack space.

- For the curious:  read about "tail recursion"
  - avoids creating stack frames in special cases

# References

*Big Java*, Chapter 13 *Recursion.*

http://codingbat.com - programming problems using recursion. First set is easy, second set is more challenging and fun.



# Recursion to Compute Permutations

*Extra slides - not required.*

This is a harder but practical example.

Recursion greatly simplifies the problem.

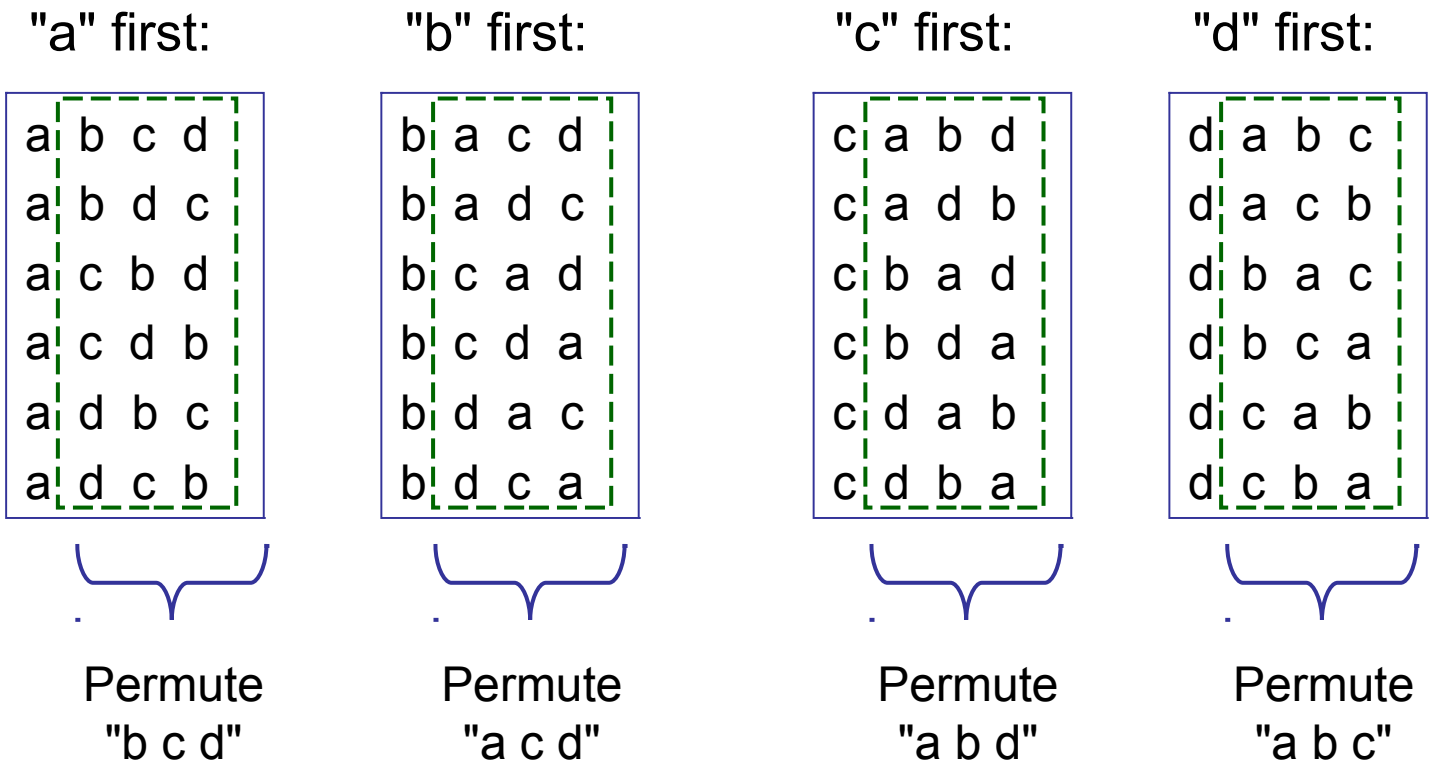
An Iterator makes it more efficient.

# Recursion to Compute Permutations

Problem: output all permutations of a group of letters.

How could you apply recursion to compute permutations?

Example: compute all permutations of "a b c d".



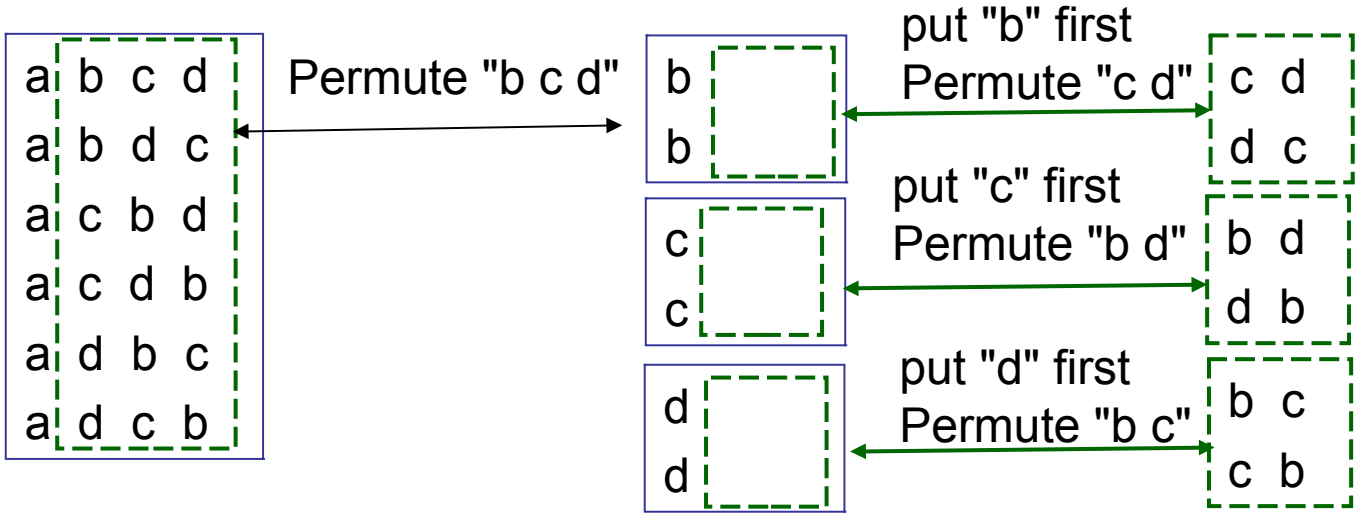
# Recursion to Compute Permutations

How could you apply recursion to compute permutations?

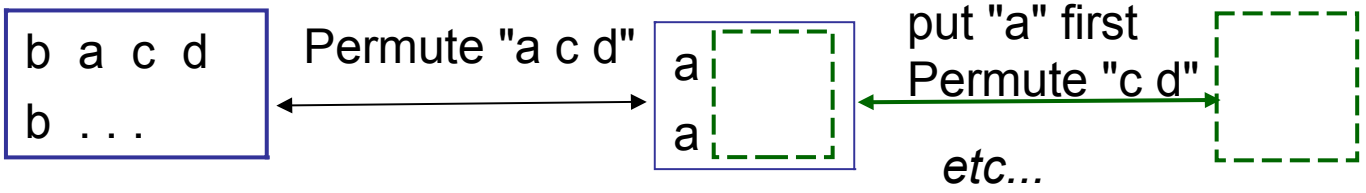
Think about how you would solve the problem yourself....

To Permute "a b c d":

1. put "a" first:

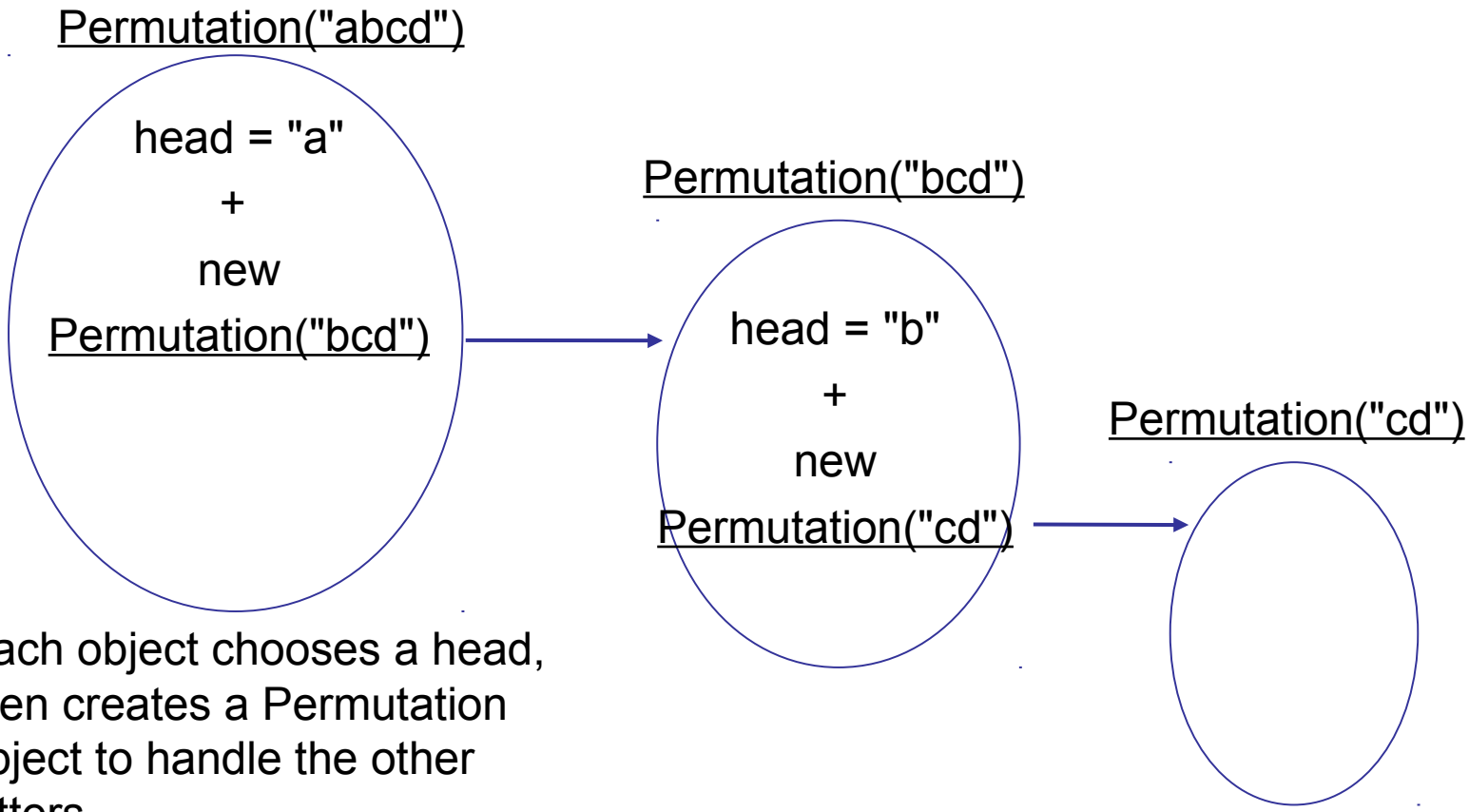



2. put "b" first:



# Designing a Permutation Class

Each Permutation object contains *another* permutation object, which it uses to compute permutations of a subnet of the data.




Each object chooses a head, then creates a Permutation object to handle the other letters.

# Special Issues for this Problem

- In many recursion problems the work is done silently.

  Example: finding a Knight Tour, counting nodes in a true.

- Other problems require output at each step

  Example: get the permutations one by one

# Defining a Permutation Class

To design a class, ask yourself:

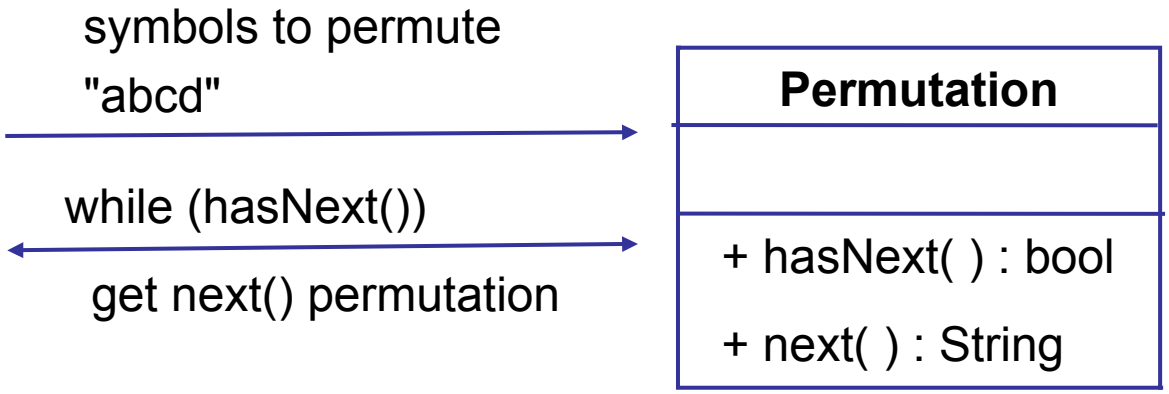
- what behavior should a Permutation object have?
- what state is necessary to perform this behavior?

There may be **many, many** permutations of letters.

Instead of computing and storing them all...

let's use an **`Iterator`**

- produce and return the permutations one at a time



# Defining a Permutation Class (2)

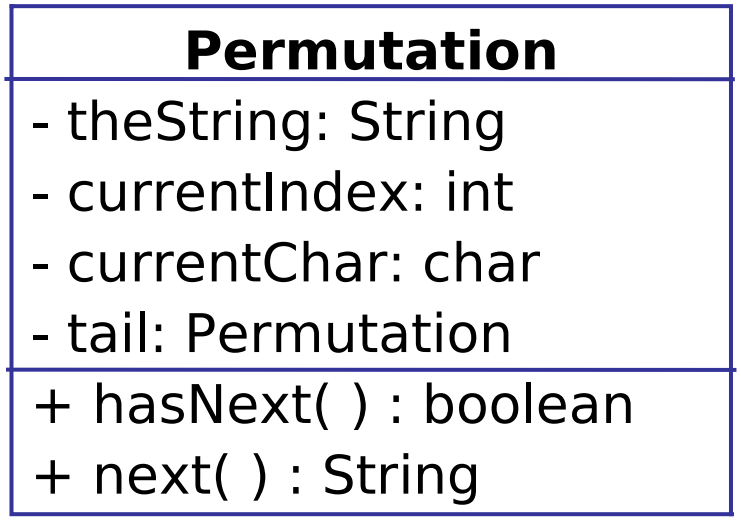
Pseudo-code for our Permutation program might look like this:

```
// create a Permutation object and tell it what to permute
Permutation me = new Permutation("abcd");

// now print all the permutations, one per line
while ( me.hasNext( ) )
    System.out.println( me.next( ) );

// wasn't that easy ?
```

# Defining a Permutation Class (3)

Now, what does a Permutation object need to know to perform this behavior? A permutation object needs...

- the chars in the String to permute
- keep track of which character has been chosen as the *first* character of permutation.
- another permutation object that is responsible to permute the remaining characters (here is where we use recursion).



# Defining a Permutation Class (4)

Using the UML class diagram, we can start to implement the Permutation class

```
/** constructor for a new Permutation.
 * @param text to permutate. Must contain at least one
 * character.
 */
public Permutation(String text) {
   if (text == null) throw new
        IllegalArgumentException("Must not be null");
    theString = text;
    currentIndex = 0;
    currentChar  = text.charAt( currentIndex );
    // create a Permutation object for recursion
    String tailString = makeTailString(currentIndex);
    tail = new Permutation(tailString);
}
```

# Method to create tailString

After we choose a character to be first element, we need to create a new String containing all the other characters.

```
/**
 * Create a new string by removing the character at
 * the given index.
 * @param index is index of char to remove from string
 * @return a new string with one character removed
 */
private String makeTailString(int index) {
    String therest = theString.substring(0, index)
    + theString.substring( index+1 );
    return therest;
}
```

# Permutation Iterator: hasNext( )

We need to define hasNext( ) to check for more permutations.

```
/** @return true if there are more permutations */
public String hasNext(  ) {
   return true if one of these is true:
   (1) tail has more permutations
   If (1) is false then...
   (2) return true if firstChar can be advanced to
       another character of the String.  In this case,
       also perform this operation.  Note that you
       must change the tail, too.
```

When you implement this, you may find some code is duplicated in hasNext( ) and next( ). Eliminate duplicate code.
Obviously hasNext() cannot call next( ) [ *why?* ],
Rewrite next( ) so that it calls hasNext( ).
The next( ) method will become very simple.

# Define the Iterator

In iterators, hasNext() usually does most of the work.

```
char c = theString.charAt( startChar );
if ( tail.hasNext() )
    return c + tail.next( );
    // returns: abcd, abdc, acbd, acdb, ...
... actually, your code is more complex than this

// if no more permutations of the tail, then we
// must choose a new startChar (and a new tail)
startChar++; // choose a new first character
char c = theString.charAt( startChar );
// construct a new tailString and a new tail object
tailString = theString.substring(0, startChar)
    + theString.substring( startChar+1 );
tail = new Permutation( tailString );
```

# Permutation Iterator: next( )

We need to define **next( )** to implement the iterator

```
/** return the next permutation of this String, if any */
public String next(  ) {
    if ( tail.hasNext( ) )
        // return firstChar followed by next
        // permutation from our tail permutation
      return startChar + tail.next( ) ;
    // otherwise, move thisChar to the next character
    firstChar++;
// TO DO: check that we have reached end of string

// TO DO: construct a new tail String
    tailString = ...;
    tail = new Permutation( tailString );
    return theString.charAt( thisChar )
           + tail.getNextPermutation( );
```

# Identify the Base Case

Base case:

1. `theString` contains 0 or 1 characters (`tailString` is empty).
2. when `currentIndex` is the last character in `theString`, what is the tail?
3.

# Ensuring Termination

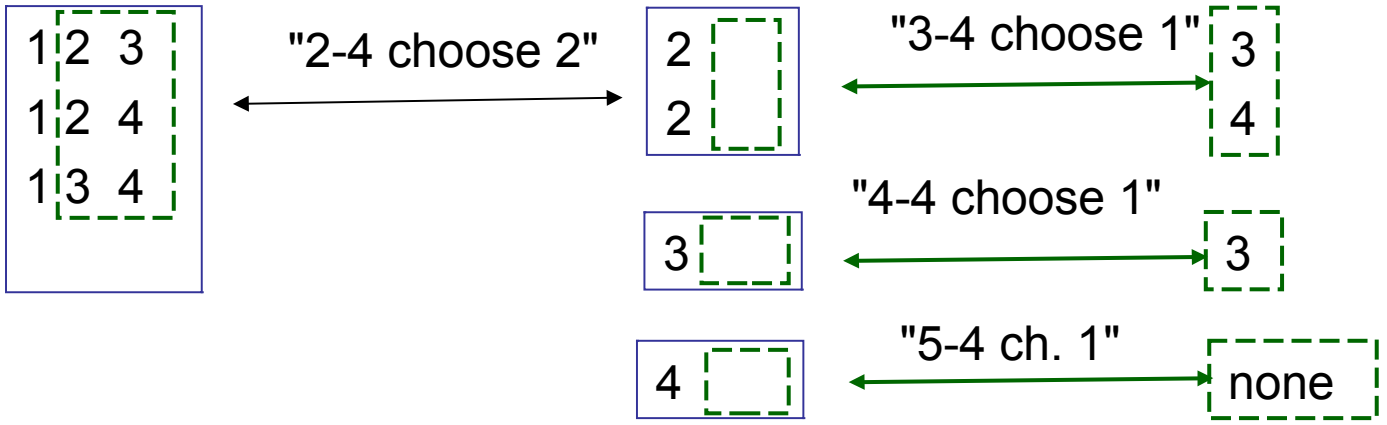
- Will this algorithm terminate?
- In the loop, hasNext( ) is called. So, will hasNext() eventually return false?
- Notice that the permutation printed at each iteration is the characters at positions this.firstChar, tail.firstChar, tail.tail.firstChar, ...
- hasNext( ) should check the value of firstChar.
- Hence, we should ensure that thisChar or tail.thisChar or tail.tail.thisChar, etc..., is incremented each time next() is used.

# Combinations

- Computing combinations of objects, say "4 choose 2" is similar to permutations.
- Recursion is *easier* for combinations.
- Example:  compute call combinations of "4 choose 3".

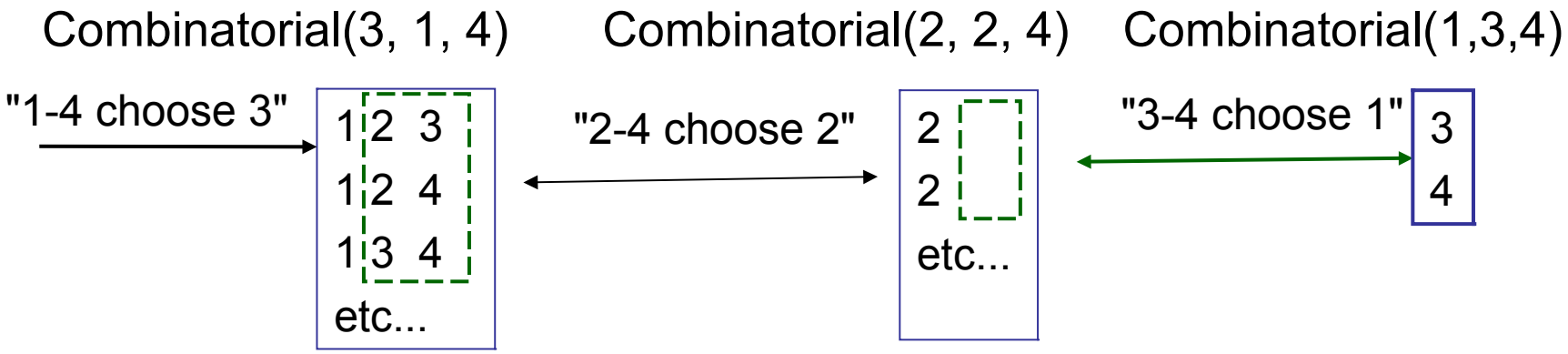
1. select "1" as first element

1.1 print all combinations of "3 choose 2" using numbers 2 - 4.



# A recursion-friendly Combinatorial

- We want to define a Combinatorial class in a way that is conducive to recursion.
- Look at the way you would solve the problem by hand.
- At each step you look for a Combinatorial of n numbers between startVal and endVal.
- Think of a constructor:
  Combinatorial( how_many, startVal, endVal )



# Combinatorial: identify behavior

- Combinatorial.hasNext( ):
  return true if there are more combinations
- Combinatorial.next( ):
  return string representation of next Combination.

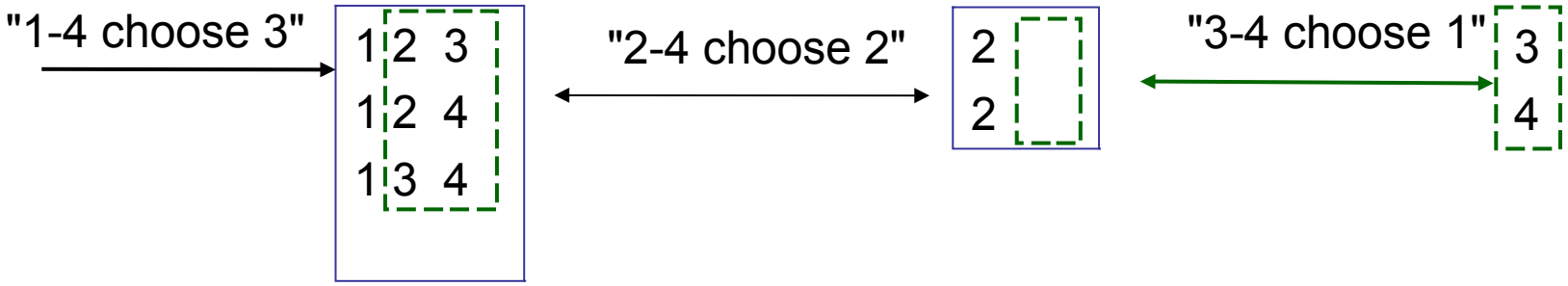
1. select "1" as first element.

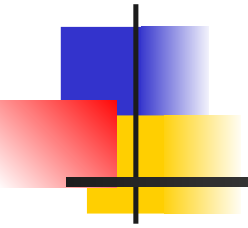
1.1 Combinatorial theRest = new Combinatorial(2, 2, 4);

1.2 while ( theRest.hasNext() )

    println "1" + theRest.next( );

2. then what?





# Knight Tour

Find a sequence of moves in chess so that a knight visits every square on chess board exactly once.

Recursion greatly simplifies the problem.

This example uses 2 special features:

(a) backtracking - undo a failed path

(b) shared state - info shared between recursive calls

# Knight Tour

- Describe the problem.
- This is an example of recursion with **backtracking**.
- For backtracking to work you need:
  - a way to go back to a previous state,
- it usually also involves a shared state.
  - "shared" data that is used by all levels of the recursion process
  - for Knight Tour, the shared information is the **chess board** containing list of occupied squares.

**Backtracking**: when an attempted solution fails, go back to a previous state (back track) and try another path to a solution

# Knight Tour

- How would you solve the knight tour problem yourself?

1. choose first move and mark it on chess board.

1.1 choose second move and mark it on chess board.

1.1.1 choose third move and mark it.

...etc...

if fails, then erase the last move(s) and try again.

...etc...

if fails, then erase 3rd move and try a different sequence.

...etc...

if fails, then erase 2nd move and try a different sequence.

# Knight Tour: identify behavior

- canMoveTo( x, y ) - true if position (x,y) is valid and you haven't visited their yet.
- moveTo( x, y ) - move to a new square and label it using the current sequence number (1, 2, 3, ...)
- unmoveTo( x, y ) - retract a move: unlabel the square
- isBoardFull( ) - test whether every square on the board has been visited (labeled) yet.
- knightTour( x, y ) - attempt to tour the chess board starting at position (x,y); any squares which have already been labeled are included in the tour. That is, you can't move to those squares again.
- knightTour( x, y ) is how we implement recursion!

# Knight Tour: implement recursion

- how to recursively traverse the board?

```
public boolean knightTour(int x, int y) {
    if ( ! canMoveTo(x,y) ) return false;  // square already visited
    moveTo(x, y);  // add this square to the knight tour
    // have we toured the entire board?
    if ( isBoardFull( ) ) return true;
    // Otherwise, try to continue the knight tour.
    // Try all 8 possible moves  (use an array and loop is better)
    if ( knightTour(x+2, y+1) ) return true; // success!
    if ( knightTour(x+1, y+2) ) return true;
    ..etc... // try all 8 possible moves.
    // if all moves fail, then unlabel this square
    unmoveTo(x, y);
    return false;  // failed
}
```

# Knight Tour: guarantee termination

Can we guarantee the recursive calls will terminate?

- at each level of recursion an additional square is added to the knight tour ( moveTo(x,y) ) or failure returned
- each new move is tried only once.
- therefore, eventually either (i) no move is possible, or (ii) all squares are occupied.

```
public boolean knightTour(int x, int y) {
    if ( ! canMoveTo(x,y) ) return false;   // FAILS is no move
    moveTo(x, y);
    // have we toured the entire board?
    if ( isBoardFull( ) ) return true;          // SUCCESS!!
    ...etc...
```

# Knight Tour: identify state

- What do we need to "remember" while trying a knight tour?
- the current position is a parameter to the knightTour( ) method, so it does *not* need to be a class attribute.
- what we need to remember is the state of the chess board -- its size and what squares have been visited.
- let theBoard[i][j] = n > 0 if (i,j) visited at move n
  = 0 if (i,j) not yet visited

```
// attributes (state variables)
private int boardSize;   // the board size
private int[][] theBoard;  // the chess board, a 2-D array
private int sequenceNum; // number of squares visited, also
                         // used to label the next square
```

# Knight Tour: build the class

```
public class ChessBoard {
        // state variables
        private int boardSize;    // the board size
        private int[][] theBoard;  // the chess board
        private int sequenceNum; // number of squares
visited
        public ChessBoard( int size ) {
                if (size < 1) /* ERROR */ System.exit(0);
                boardSize = size;
                theBoard = new int[size][size];
                sequenceNum = 0;
                // set all elements of theBoard[ ][ ] to 0.
        }
        public boolean knightTour(int x, int y) { ... }
        public boolean canMoveTo(int x, int y) { ... }
        public void moveTo(int x, int y) { ... }
        public void unmoveTo(int x, int y) { theBoard(x,y) =
0; }
```

# Knight Tour: starting the tour

- perform the knight tour using the main( ) method.
- create a ChessBoard object and call knightTour( )

```
public class ChessBoard {
    // define attributes (state variables)
    ...
    // define methods
    ...
    public static void main( String [] args ) {
        ChessBoard board = new ChessBoard(5);
        // start at a corner
        if ( board.knightTour( 0, 0 ) )
        {   System.out.println("Found a tour!");
            board.printBoard( );
        }
```

# Self-sequencing the Knight Tour

- the ChessBoard object keeps track of sequence numbers so you don't have to
- initially, sequenceNum = 0 (no moves yet)
- each time moveTo( ) a square, increment sequenceNum
- each time unmoveTo( ) a square, decrease sequenceNum

```
public void moveTo(int x, int y) {
    sequenceNum++;
    theBoard[x][y] = sequenceNum;
}
public void unmoveTo(int x, int y) {
    sequenceNum--;
    theBoard[x][y] = 0;
}
```

# Checking for feasible moves

- canMoveTo(x, y) checks for feasible moves. Doing bounds checking here makes the rest of the program simpler.

```
public boolean canMoveTo(int x, int y) {
    // bounds checking
    if ( x < 0 || x >= boardSize ) return false;
    if ( y < 0 || y >= boardSize ) return false;
    // return true if board at (x,y) is unoccupied (0)
    return theBoard[x][y] == 0 ;
}
```

# KnightTour: modification for efficiency

- canMoveTo( ) and moveTo( ) are always called as a pair

```
public boolean knightTour(int x, int y) {
    if ( ! canMoveTo(x,y) ) return false;  // square already visited
    moveTo(x, y);  // add this square to the knight tour
    ...etc...
```

- to reduce function calls and make the program faster, combine canMoveTo( ) and moveTo( ) into one: moveTo(x,y) returns true if a move succeeds

```
public boolean knightTour(int x, int y) {
    if ( ! moveTo(x,y) ) return false;  // square already visited
    ...etc...
```