Object-Oriented Data Structures Using Java

Chapter 7

Programming with Recursion

What Is Recursion?

- **Recursive call** A method call in which the method being called is the same as the one making the call
- **Direct recursion** Recursion in which a method directly calls itself
- **Indirect recursion** Recursion in which a chain of two or more method calls returns to the method that originated the chain

Recursion

- You must be careful when using recursion.
- Recursive solutions can be less efficient than iterative solutions.
- Still, many problems lend themselves to simple, elegant, recursive solutions.

Some Definitions

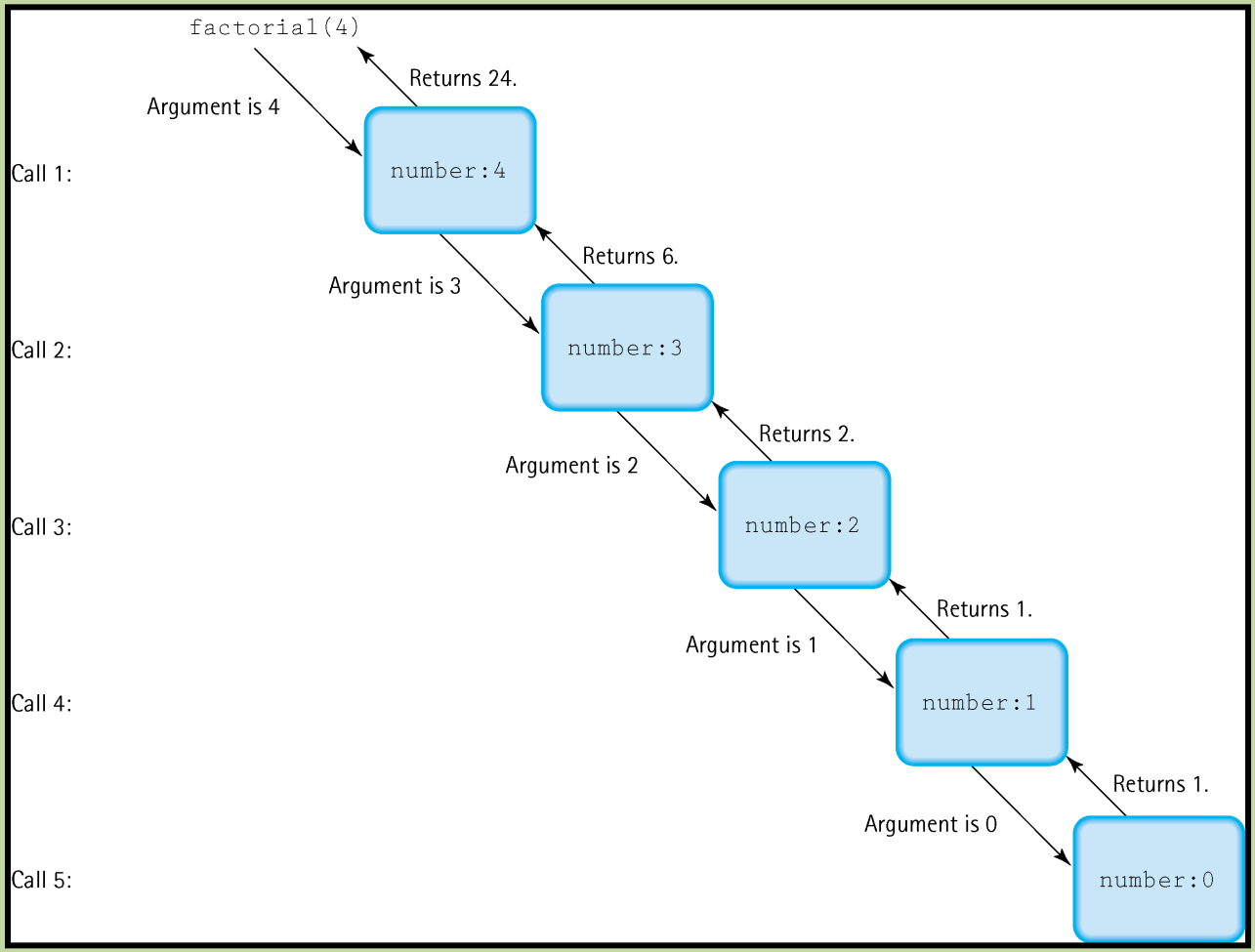
- **Base case** The case for which the solution can be stated nonrecursively
- **General (recursive) case** The case for which the solution is expressed in terms of a smaller version of itself
- **Recursive algorithm** A solution that is expressed in terms of (a) smaller instances of itself and (b) a base case

The Factorial Function

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1) * (n-2) * \cdots * 1, & \text{if } n > 0 \end{cases}$$

```
public static int factorial(int number)
{
  if (number == 0)
    return 1;                  // Base case
  else
    return (number * factorial(number - 1));      // General case
}
```

Execution of `factorial (4)`



The Factorial Algorithm

- Is familiar
- Easy to visualize
- In practice, one would never want to solve this problem using recursion.
- The iterative solution is simpler and much more efficient.

// iterative solution

```
// Iterative solution
public static int factorial(int number)
{
  int value = 1;
  for (int count = 2; count <= number; count++)
  {
    value = value * count;
  }
  return value;
}
```

Verifying Recursive Methods

The Three-Question Method

1. *The Base-Case Question:* Is there a nonrecursive way out of the method, and does the method work correctly for this base case?
2. *The Smaller-Caller Question:* Does each recursive call to the method involve a smaller case of the individual problem, leading inescapably to the base case?
3. *The General-Case Question:* If the recursive call(s) works correctly, does the whole method work correctly?

Writing Recursive Methods

1. Get an exact definition of the problem to be solved.
2. Determine the size of the problem to be solved on this call to the method.
3. Identify and solve the base case(s) in which the problem can be expressed nonrecursively.
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem—a recursive call.

Recursive isThere

Recursive isThere(item) method: returns boolean

Definition:	Searches list for item. Returns true if item is found; returns false otherwise.
Size:	The number of slots to search in list.info[startPosition]..list.info[list.numItems − 1].
Base Cases:	(1) If list.info[startPosition] equals value, return true.
	(2) If startPosition equals list.numItems − 1 and list.info[list.length − 1] does not equal item, return false.
General Case:	Search the rest of the list for item. This is a recursive invocation of isThere with a parameter startPosition + 1 (smaller caller).

The code for the recursive isThere

```
public boolean isThere (Listable item, int startPosition)
// Returns true if item is on this list; otherwise, returns false
{
  if (item.compareTo(list[startPosition]) == 0)  // If they match
    return true;
  else if (startPosition == (numItems - 1))      // If end of list
    return false;
  else return isThere(item, startPosition + 1);
}
```

How many combinations of a certain size can be made out of a total group of elements?

For instance, if we have 20 different books to pass out to four students, we can easily see that—to be equitable—we should give each student five books. But, how many combinations of five books can be made out of a group of 20 books?

Recursive Mathematical Formula

$$C(group, members) = \begin{cases} group, & \text{if } members = 1 \\ 1, & \text{if } members = group \\ C(group - 1, members - 1) + C(group - 1, members), & \text{if } group > members > 1 \end{cases}$$

Summarize our problem

Number of Combinations, returns int

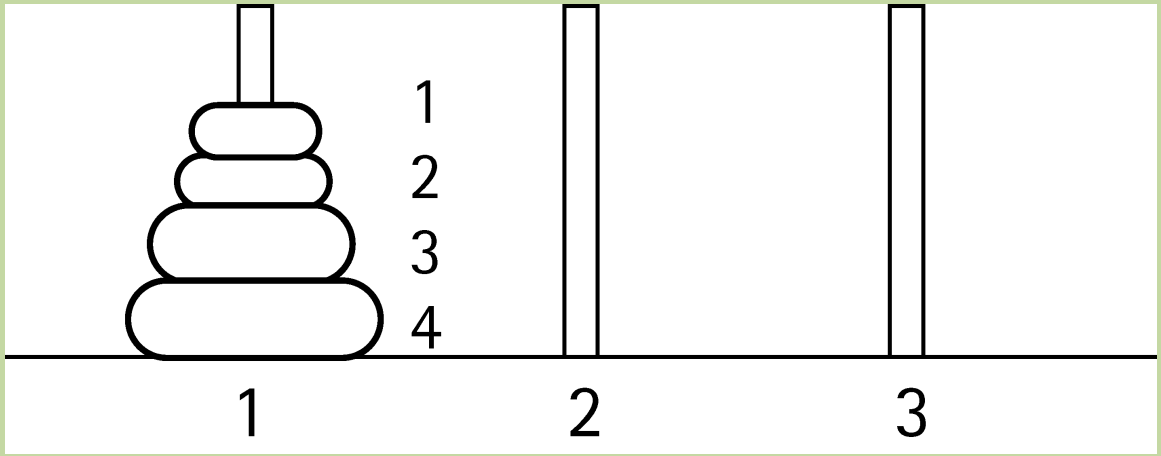
Definition:	size can be made from the total group size.
Size:	Sizes of group, members.
Base Case:	(1) If members = 1, return group.
	(2) If members = group, return 1.
General Case:	If group > members > 1, return
	Combinations(group − 1, members − 1) +
	Combinations(group − 1, members)

The resulting recursive method, `combinations`

```
public static int combinations(int group, int members)
// Pre:  group and members are positive
// Post: Return value = number of combinations of members size
//       that can be constructed from the total group size
{
  if (members == 1)
    return group;                  // Base case 1
  else if (members == group)
    return 1;                      // Base case 2
  else
    return (combinations(group - 1, members - 1) +
            combinations(group - 1, members));
}
```

Towers of Hanoi

The object of the game is to move the circles, one at a time, to the third peg. The catch is that a circle cannot be placed on top of one of that is smaller in diameter.



The General Algorithm

Get n Circles Moved from Peg 1 to Peg 3

Get n − 1 circles moved from peg 1 to peg 2
Move the nth circle from peg 1 to peg 3
Get n − 1 circles moved from peg 2 to peg 3

The Program

```
public static void doTowers(
   int circleCount,     // Number of circles to move
   int beginPeg,        // Peg containing circles to move
   int auxPeg,          // Peg holding circles temporarily
   int endPeg       )   // Peg receiving circles being moved
// Moves are written on file outFile
{
  if (circleCount > 0)
  {
    // Move n - 1 circles from beginning peg to auxiliary peg
    doTowers(circleCount - 1, beginPeg, endPeg, auxPeg);

    outFile.println("Move circle from peg " + beginPeg
              + " to peg " + endPeg);

    // Move n - 1 circles from auxiliary peg to ending peg
    doTowers(circleCount - 1, auxPeg, beginPeg, endPeg);
  }
}
```

A Recursive Version of Binary Search

binarySearch (item, fromLocation, toLocation), returns boolean

Definition:	Searches the list delimited by the parameters to see if item is present.
Size:	The number of elements in list[fromLocation] ... list[toLocation].
Base Cases:	(1) If fromLocation > toLocation, return false.
	(2) If item.compareTo(list[midPoint]) = 0, return true.
General Case:	If item.compareTo(list[midPoint]) < 0, binarySearch the first half of the list.
	If item.compareTo(list[midPoint]) > 0, binarySearch the second half of the list.

The Method

```
private boolean binarySearch (Listable item, int fromLocation, int toLocation)
// Returns true if item is on this list, between fromLocation and toLocation;
//  otherwise, returns false
{
  if (fromLocation > toLocation)           // Base case 1
    return false;
  else
  {
    int midPoint;
    int compareResult;
    midPoint = (fromLocation + toLocation) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)              // Item found
      return true;
    else if (compareResult < 0)
      // Item is less than element at location
      return binarySearch (item, fromLocation, midPoint - 1);
    else
      // Item is greater than element at location
      return binarySearch (item, midPoint + 1, toLocation);
  }
}
```

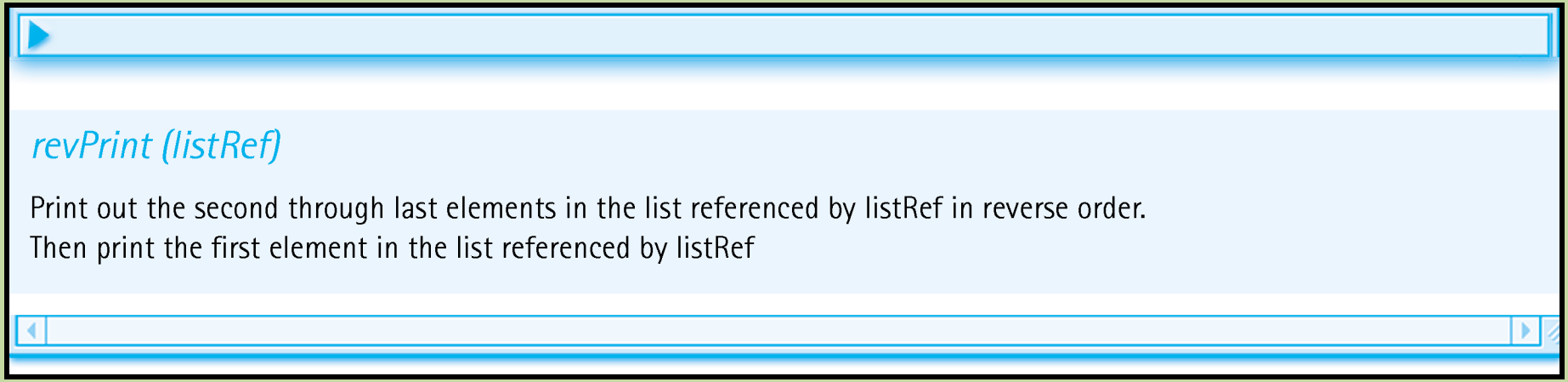
The `isThere` Method

- Now consists of a single call to the recursive method, passing it the original values for `fromLocation` and

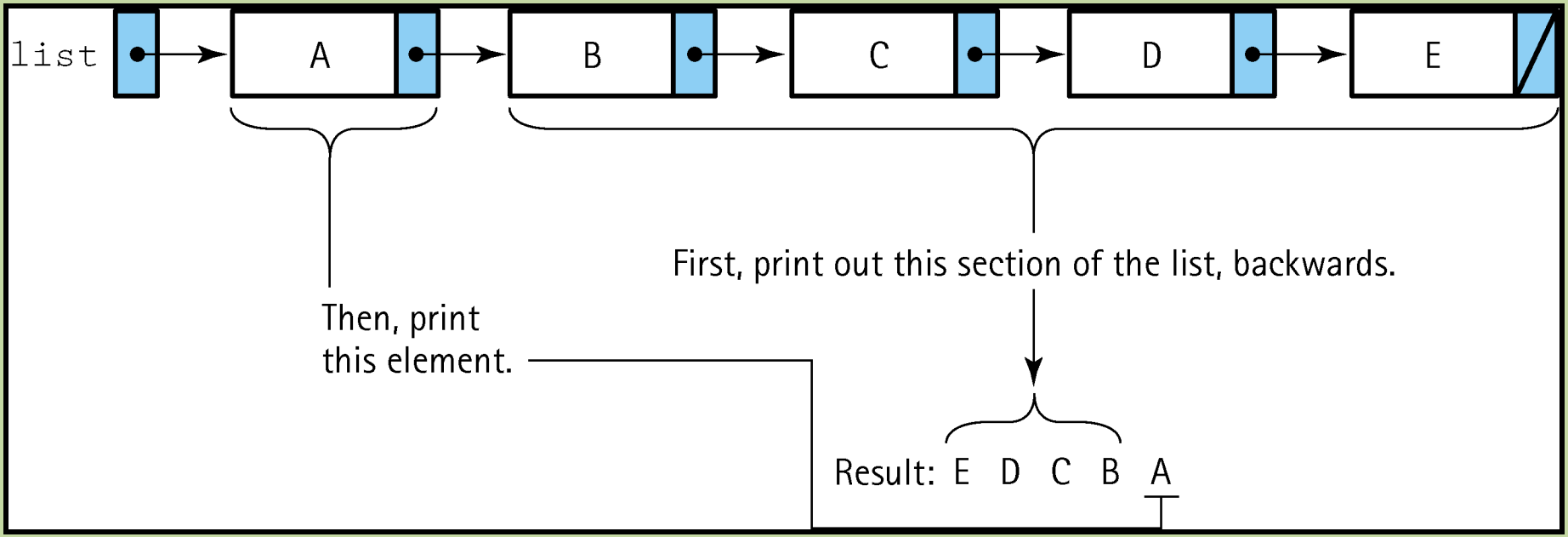
```
public boolean isThere (Listable item)
// Returns true if item is on this list; otherwise, returns false
{
  return binarySearch(item, 0, numItems - 1);
}
```

Reverse Printing

- Print out the elements of a linked list in reverse order.



Recursive `revPrint`



The Cases

Reverse Print (listRef)

Definition:	Prints out the list referenced by listRef in reverse order.
Size:	Number of elements in the list referenced by listRef.
Base Case:	If the list is empty, do nothing.
General Case:	Reverse Print the list referenced by listRef.next, then print listRef.info.

The Code

- We must create `revPrint` as an auxiliary, private method and define a public method, say, `PrintReversed`, which calls `revPrint`:

Recursive insertion into a sorted linked list

- Definition: Insert item into the sorted list referenced by subList.
- Size: The number of items in subList.
- Base Case: (1) If subList is empty, insert item into the empty list, and return this new list. (2) If item is less than first element on subList insert item at the beginning of subList, and return this new list.
- General Case: Set subList to the list returned by recursive Inser(subList.next, item) and return subList.

The Algorithm

recursiveInsert (subList, item): Returns List

```
if subList is empty
    return a list consisting of just item
else if item is less than the first item on subList
    insert item onto the front of subList
    return this new list
else
    subList.next = recursiveInsert(subList.next, item)
    return subList
```

The Code

Using `recursiveInsert`

It is good to use recursion when:

- The depth of recursive calls is relatively "shallow."
- The recursive version does about the same amount of work as the nonrecursive version.
- The recursive version is shorter and simpler than the nonrecursive version.