JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris August 2001

15 RECURSION

15.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- explain the principles of recursion
- trace a sequence of recursive method calls
- write and test simple recursive methods

15.2 PRE-REQUISITES

The student should be comfortable with using arrays and the linear search, binary search and selection sort methods.

15.3 INTRODUCTION

We explore principles of recursion through worked examples.

15.4 POWER

We have probably worked out 2^5 something like this:

We started with the simplest case: $2^{\mathbf{0}} = 1$

Then we multiplied the previous result by 2: $2^{\mathbf{1}} = 2 \times 2^{\mathbf{0}} = 2$

Then we multiplied the previous result by 2: $2^{\mathbf{2}} = 2 \times 2^{\mathbf{1}} = 4$

Then we multiplied the previous result by 2: $2^{\mathbf{3}} = 2 \times 2^{\mathbf{2}} = 8$

Then we multiplied the previous result by 2: $2^{\mathbf{4}} = 2 \times 2^{\mathbf{3}} = 16$

Then we multiplied the previous result by 2: $2^{\mathbf{5}} = 2 \times 2^{\mathbf{4}} = 32$

In general, $2^{\mathbf{n}} = 2 \times 2^{\mathbf{n-1}}$
provided *n* is an integer >= 0.

Now we can write the Java method to return the value of two raised to some power, *n*.

```
public static int powerOfTwo(int n)
/* returns two raised to some power n.
   Requires n >= 0.
*/
{
  if (n == 0)                  // the simple case
    return 1;
  else if (n > 0)              // the recursive step,
    return 2 * powerOfTwo(n-1); // one step closer to the
                               // simple case
  return -1;                   // error, n cannot be negative
}
```

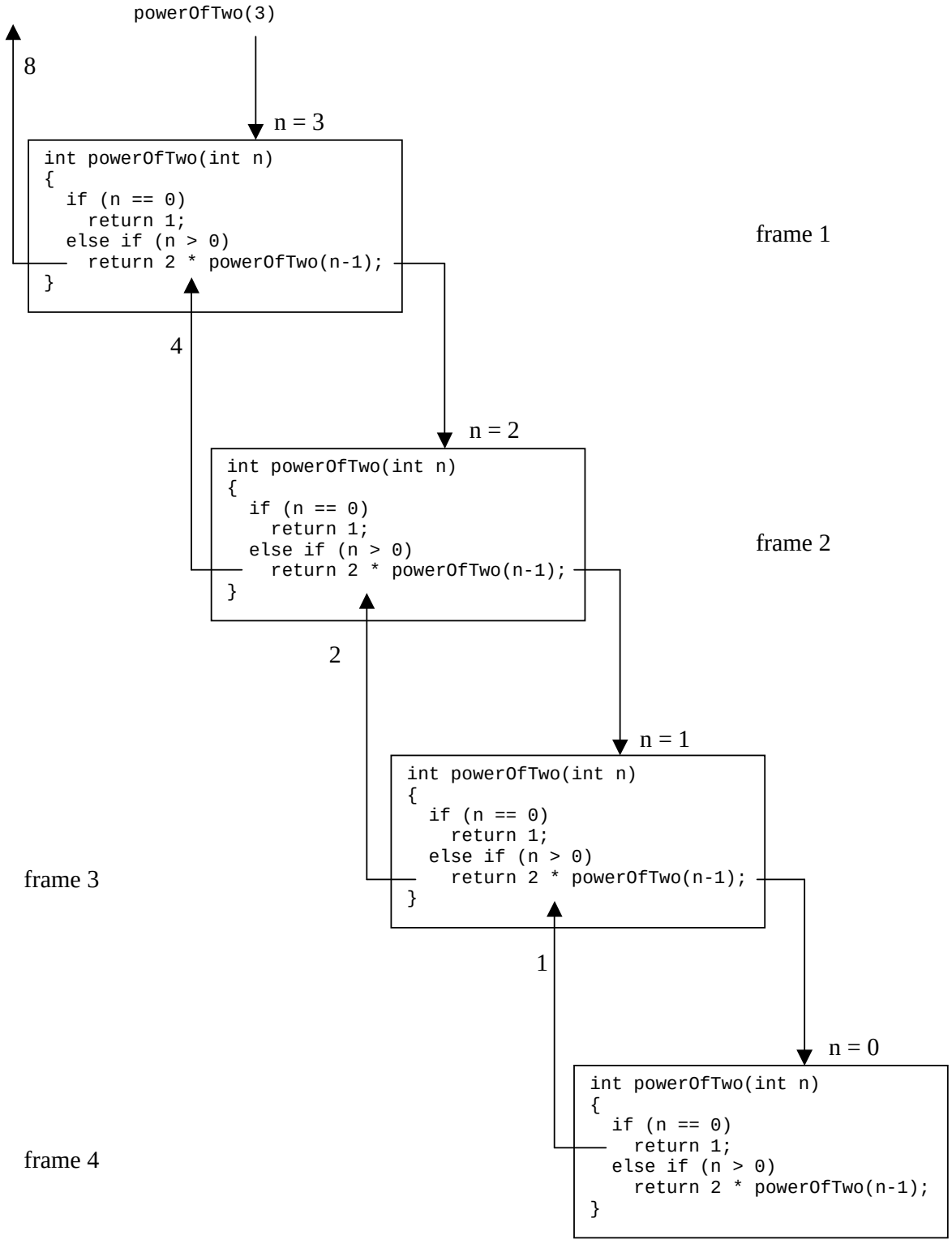
`System.out.println(powerOfTwo(3));` results in *8* being displayed.

A recursive method is one that calls itself. The *powerOfTwo(int n)* method described above makes a call to itself in *powerOfTwo(n-1)*.

In the simplest form of recursion, we always have

(a) the simple case that ends the sequence of recursive calls and
(b) the recursive step, that is always one step closer to the simple case

Just to emphasise the principles, let us trace the sequence of method calls resulting from *System.out.println(powerOfTwo(3));*.


powerOfTwo(3)
8
n = 3
int powerOfTwo(int n)
{
if (n == 0)
return 1;
else if (n > 0)
return 2 * powerOfTwo(n-1);
}
frame 1
4
n = 2
int powerOfTwo(int n)
{
if (n == 0)
return 1;
else if (n > 0)
return 2 * powerOfTwo(n-1);
}
frame 2
2
n = 1
int powerOfTwo(int n)
{
if (n == 0)
return 1;
else if (n > 0)
return 2 * powerOfTwo(n-1);
}
frame 3
1
n = 0
int powerOfTwo(int n)
{
if (n == 0)
return 1;
else if (n > 0)
return 2 * powerOfTwo(n-1);
}
frame 4

powerOfTwo(3) makes the first call to *int powerofTwo(int)* in frame 1.

In frame 1, *n* = 3. Since *n* > 0 the recursive call *powerOfTwo(2)* is made.

In frame 2, *n* = 2. Since *n* > 0 the recursive call *powerOfTwo(1)* is made.

In frame 3, *n* = 1. Since *n* > 0 the recursive call *powerOfTwo(0)* is made.

In frame 4, *n* = 0. So **1** is returned to frame 3.

In frame 3, *powerOfTwo(0)* is **1**. So 2 * 1 = **2** is returned to frame 2.

In frame 2, *powerOfTwo(1)* is **2**. So 2 * 2 = **4** is returned to frame 1.

In frame 1, *powerOfTwo(2)* is **4**. So 2 * 4 = **8** is returned.

In general, calculation and return is suspended until the result from the next recursive call is obtained.

Recursion is a way of thinking.

15.5 LINEAR SEARCH

The method *contains()* shown below performs a recursive linear search on an array of *int*s for a given *int*, *target*. It searches through the array from the given *fromIndex* up to the given *toIndex* inclusive.

The simple case occurs when *fromIndex* exceeds *toIndex*; then *target* is not in the array.

We check the first item in the array:

```
array[fromIndex] == target
```

before going on to search the rest of the array with the recursive call *contains(array, fromIndex+1, toIndex, target)*. The recursive call brings you one step closer to the simple case. Since each call adds 1 to *fromIndex* there must come a time when *fromIndex* exceeds *toIndex*.

```
public static boolean contains(int[] array,
                               int fromIndex, int toIndex,
                               int target)
{
  if (fromIndex > toIndex)
    return false;
  return array[fromIndex] == target ||
            contains(array, fromIndex+1, toIndex, target);
}
```

Given the array

```
int[] array = { 3, 5, 7, 11, 13 };
```

the call to *contains(array, 0, 4, 7)* results in

false || *false* || *true* || *false* || *false* being evaluated. The end result, of course, is *true*.

15.6 BINARY SEARCH

The *binarySearch()* method shown below is implemented using recursion. The method sorts the given array of *int*s between the given indices, *lo* and *hi*.

There are two simple cases that terminate the sequence of recursive calls.

If *lo* exceeds *hi* then there is nothing left to search and the *target*, the *int* we are looking for, is not in the array.

If the *target* is in the *middle* location of the array, then we have found what we were looking for.

There are two possible recursive calls, each one a step closer to a simple case.

binarySearch(array, lo, middle-1, target) searches the left-hand part of the array.

binarySearch(array, middle+1, hi, target) searches the right hand part of the array.

```
public static boolean binarySearch(
              int[] array, int lo, int hi, int target)
{
  if (lo > hi)
    return false;
  int middle = (lo + hi) / 2;
  if (array[middle] == target)
    return true;
  else if (target < array[middle])
    return binarySearch(array, lo, middle-1, target);
  return binarySearch(array, middle+1, hi, target);
}
```

15.7 SELECTION SORT

The selection sort method shown below is implemented using recursion.

We have two helper methods. *swap()* exchanges the values in the array at two given indices, *i* and *j*. *minLocation()* returns the location of the least value in the give array between the two indices *from* and *to* inclusive.

The simple case occurs when *from* exceeds *to*; there is no more array to sort.

Each recursive call brings you one step closer to the simple case. For each recursive call made, *from* is advanced by one so there is successively smaller and smaller portions of the array from which to select the least value and place it in its correct position.

```
public static void swap(int[] array, int i, int j)
{
  int t = array[i];
  array[i] = array[j];
  array[j] = t;
}


public static int minLocation(int[] array, int from, int to)
{
  int min = Integer.MAX_VALUE;
  int minLoc = -1;
  for (int i = from; i <= to; i++) {
    if (array[i] < min) {
      min = array[i];
      minLoc = i;
    }
  }
  return minLoc;
}


public static void selectionSort(
                               int[] array, int from, int to)
{
  if (from > to)
    return;
  int minLoc = minLocation(array, from, to);
  swap(array, from, minLoc);
  selectionSort(array, from+1, to);
}
```

15.8 REVIEW

15.9 FURTHER READING

ARNOW D & WEISS G *Introduction to Programming Using Java* pp 461
LEWIS & LOFTUS *Java Software Solution* pp 468

In the next lesson we use recursion to implement binary search tree operations.

15.10 EXERCISES

1 Trace the sequence of recursive method calls for method Q shown below when the first call to method Q is *System.out.println(Q(5));*

```
public static int Q(int n)
{
  if (n == 1)
    return 1;
  return 13 + Q(n - 1);
```

```
}
```

2 Trace the sequence of recursive method calls for method X shown below when the first call to method X is *System.out.println(sum(2, 3));* Hence (or otherwise) give a descriptive name to X.

```
public static int X(int m, int n)
{
  if (n == 0)
    return m;
  if (n == 1)
    return m + 1;
  return X(m+1, n-1);
}
```

3 Trace the sequence of recursive method calls for method Y shown below for each of

(a) System.out.println(Y(13, 7));
(b) System.out.println(Y(14, 7));
(c) System.out.println(Y(15, 7));

Hence (or otherwise) give a descriptive name to Y.

```
public static int Y(int m, int n)
{
  if (m < n)
    return m;
  return Y(m-n, n);
}
```

4 Develop and test a method that uses recursion to determine the result of m^n where m and n are both integers >= 0.

5 Test the recursive linear search method defined in §15.5 above. Remember to carefully check boundaries.

6 Thoroughly test the recursive binary search method defined in §15.6 above (see §5.5). Which implementation, the iterative one described in §5 .4 or the recursive one, do you find easier to understand? Explain why.

7 Test the recursive selection sort method described in §15.7 above.