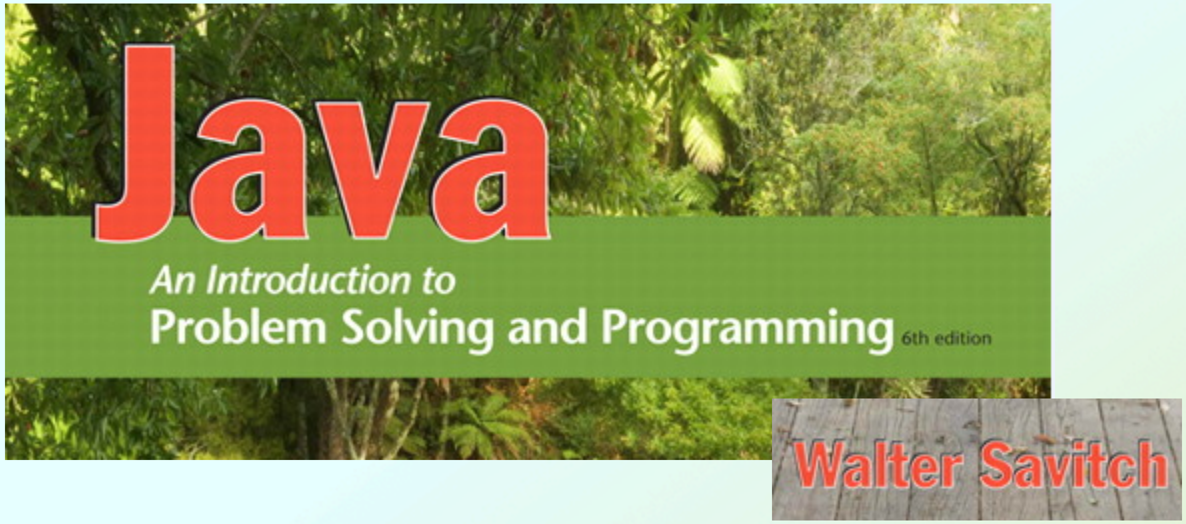


Recursion

Chapter 11

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

Objectives

- Describe the concept of recursion
- Use recursion as a programming tool
- Describe and use recursive form of binary search algorithm
- Describe and use merge sort algorithm

Basics of Recursion: Outline

- Basics of Recursion
- Case Study: Digits to Words
- How Recursion Works
- Infinite Recursion
- Recursive versus Iterative Methods
- Recursive Methods that Return a Value

Basics of Recursion

- A recursive algorithm will have one subtask that is a small version of the entire algorithm's task
- A recursive algorithm contains an invocation of itself
- Must be defined correctly else algorithm could call itself forever or not at all

Simple Example - Countdown

- Given an integer value *num* output all the numbers from *num* down to 1
- Can do this easier and faster with a loop; the recursive version is an example only
- First handle the simplest case; the **base case** or stopping condition

```
public static void countDown(int num)
{
    if (num <= 0)
    {
        System.out.println();
    }
}
```

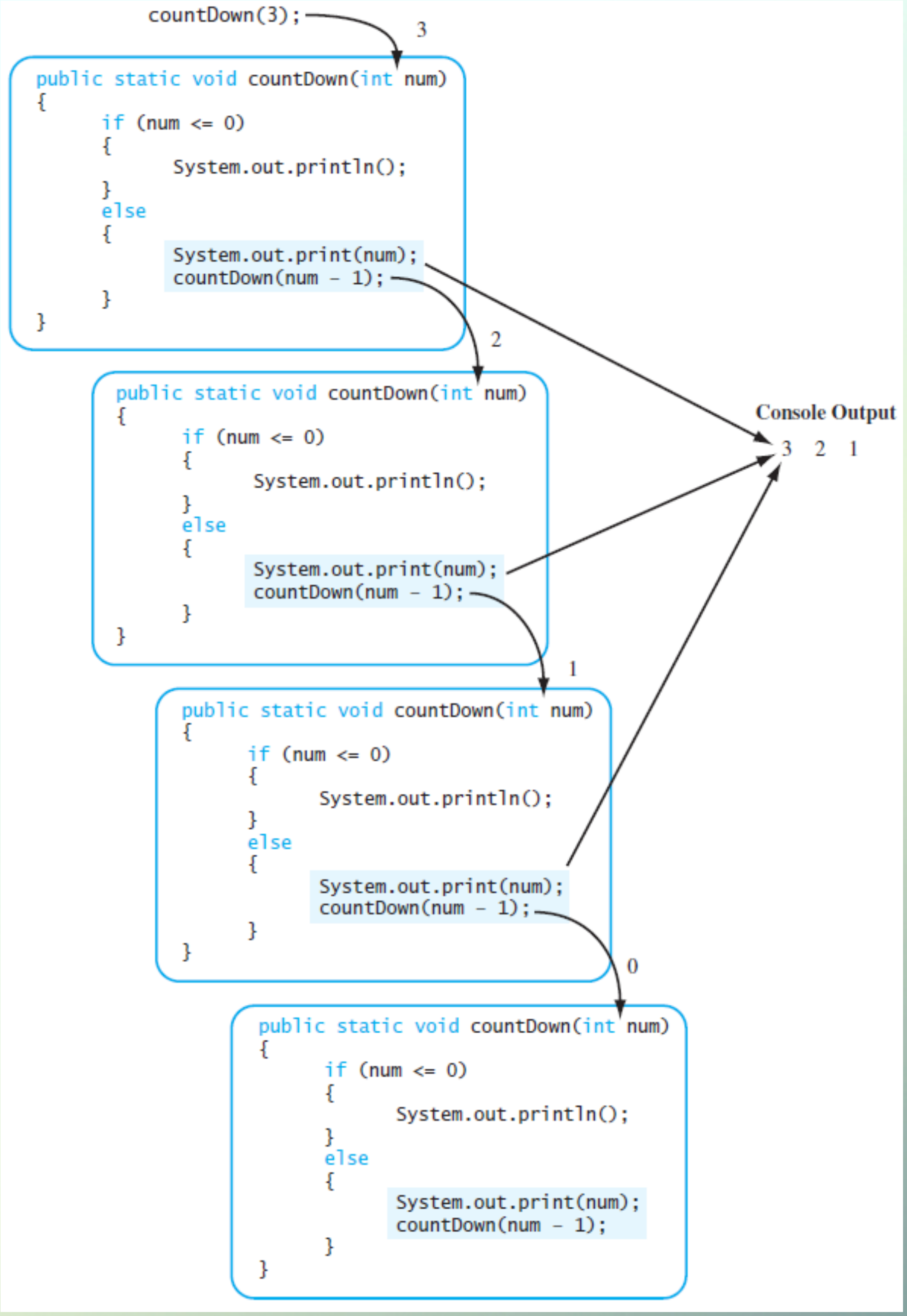
Recursive Countdown

- Next handle larger cases; phrase solution in terms of a smaller version of the same problem
- `countDown(3)` is to output 3 then output the result of `countDown(2)`

View demonstration, listing 11.1
`class RecursionCountdown`

Sequence of Calls

`countDown(3)`



JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709

Case Study

- Digits to Words – consider a method which receives an integer parameter
  - Then it prints the digits of the number as words
- Heading

```
/**
 Precondition: number >= 0
 Displays the digits in number as words.
*/
public static void displayAsWords(int number)
```

Case Study

- Consider this useful private method

```
// Precondition: 0 <= digit <= 9
// Returns the word for the argument digit.
private static String getWordFromDigit(int digit)
```

Case Study

- If number has multiple digits, decompose algorithm into two subtasks
  1. Display all digits but the last as words
  2. Display last digit as a word
- First subtask is smaller version of original problem
  - Same as original task, one less digit

Case Study

- Algorithm for **`displayAsWords(number)`**

1. **`displayAsWords`** (number after deleting last digits)
2. **`System.out.print`** (**`getWordFromDigit`**(last digit of number + " ")

Case Study

- View demonstration, listing 11.2

  `class RecursionDemo`

Sample screen output

```
Enter an integer:
987
The digits in that number are:
nine eight seven
If you add ten to that number,
the digits in the new number are:
nine nine seven
```

How Recursion Works

- Figure 11.2a Executing recursive call

`displayAsWords(987)` is equivalent to executing:

```
{//Code for invocation of displayAsWords(987)
    if (987 < 10)
        System.out.print(getWordFromDigit(987) + " ");
    else //987 has two or more digits
    {
        displayAsWords(987 / 10);
        System.out.print(getWordFromDigit(987 % 10) + " ");
    }
}
```

Computation waits here for the completion of the recursive call. (pointing to `displayAsWords(987 / 10);`)

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

How Recursion Works

- Figure 11.2b Executing recursive call

`displayAsWords(987/10)` is equivalent to `displayAsWords(98)`, which is equivalent to executing:

```
{//Code for invocation of displayAsWords(98)
    if (98 < 10)
        System.out.print(getWordFromDigit(98) + " ");
    else //98 has two or more digits
    {
        displayAsWords(98 / 10);
        System.out.print(getWordFromDigit(98 % 10) + " ");
    }
}
```

Computation waits here for the completion of the recursive call.

How Recursion Works

- Figure 11.2c Executing recursive call

`displayAsWords(98/10)` is equivalent to `displayAsWords(9)`, which is equivalent to executing:

```
{//Code for invocation of displayAsWords(9)
    if (9 < 10)
        System.out.print(getWordFromDigit(9) + " ");
    else //9 has two or more digits
    {
        displayAsWords(9 / 10);
        System.out.print(getWordFromDigit(9 % 10) + " ");
    }
}
```

Another recursive call does not occur.

Keys to Successful Recursion

- Must have a branching statement that leads to different cases
- One or more of the branches should have a recursive call of the method
  - Recursive call must us "smaller" version of the original argument
- One or more branches must include *no* recursive call
  - This is the base or stopping case

Infinite Recursion

- Suppose we leave out the stopping case

```
public static void displayAsWords(int number)//Not quite right
{
    displayAsWords(number / 10);
    System.out.print(getWordFromDigit(number % 10) + " ");
}
```

- Nothing stops the method from repeatedly invoking itself
  - Program will eventually crash when computer exhausts its resources (stack overflow)

Recursive Versus Iterative

- Any method including a recursive call can be rewritten
  - To do the same task
  - Done *without* recursion
- Non recursive algorithm uses *iteration*
  - Method which implements is *iterative method*
- Note iterative version of program, listing 11.3
  `class IterativeDemo`

Recursive Versus Iterative

- Recursive method
  - Uses more storage space than iterative version
  - Due to overhead during runtime
  - Also runs slower
- However in *some* programming tasks, recursion is a better choice, a more elegant solution

Recursive Methods that Return a Value

- Follow same design guidelines as stated previously
- Second guideline also states
  - One or more branches includes recursive invocation *that leads to the returned value*
- View program with recursive value returning method, listing 11.4
  `class RecursionDemo2`

Recursive Methods that Return a Value

Sample screen output

```
Enter a nonnegative number:
2008
2008 contains 2 zeros.
```

- Note recursive method **`NumberOfZeros`**
  - Has two recursive calls
  - Each returns value assigned to **`result`**
  - Variable **`result`** is what is returned

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

Programming with Recursion: Outline

- Programming Example: Insisting that User Input Be Correct
- Case Study: Binary Search
- Programming Example: Merge Sort – A Recursive Sorting Method

Programming Example

- Insisting that user input be correct
  - Program asks for a input in specific range
  - Recursive method makes sure of this range
  - Method recursively invokes itself as many times as user gives incorrect input
    - Dangerous technique – can result in stack overflow if invalid entries entered repeatedly
- View program, listing 11.5

```
class CountDown
```

Programming Example

Sample screen output

```
Enter a positive integer:
0
Input must be positive.
Try again.
Enter a positive integer:
3
Counting down:
3, 2, 1, 0, Blast Off!
```

Case Study

- Binary Search
  - We design a recursive method to tell whether or not a given number is in an array
  - Algorithm assumes array is sorted
- First we look in the middle of the array
  - Then look in first half or last half, depending on value found in middle

Binary Search

- Draft 1 of algorithm

```
m = an index between 0 and (a.length - 1)
if (target == a[m])
    return m;
else if (target < a[m])
    return the result of searching a[0] through a[m - 1]
else if (target > a[m])
    return the result of searching a[m + 1] through a[a.length - 1]
```

  - Algorithm requires additional parameters

JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709

Binary Search

- Draft 2 of algorithm to search **a[first]** through **a[last]**

```
mid = approximate midpoint between first and last
if (target == a[mid])
      return mid
else if (target < a[mid])
      return the result of searching a[first] through a[mid - 1]
else if (target > a[mid])
      return the result of searching a[mid + 1] through a[last]
```

  - What if target is not in the array?

JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709

Binary Search

- Final draft of algorithm to search **`a[first]`** through **`a[last]`** to find **`target`**

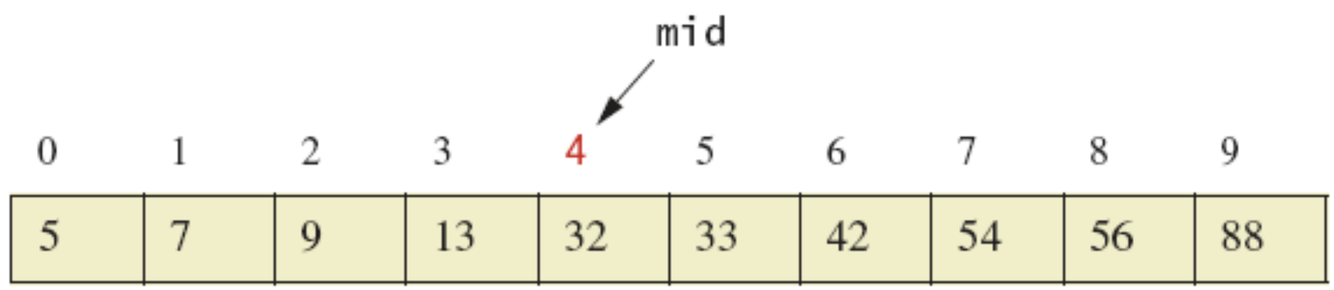
```
mid = approximate midpoint between first and last
if (first > last)
    return -1
else if (target == a[mid])
    return mid
else if (target < a[mid])
    return the result of searching a[first] through a[mid – 1]
else if (target > a[mid])
    return the result of searching a[mid + 1] through a[last]
```

Binary Search

- Figure 11.3a Binary search example

`target` *is* 33

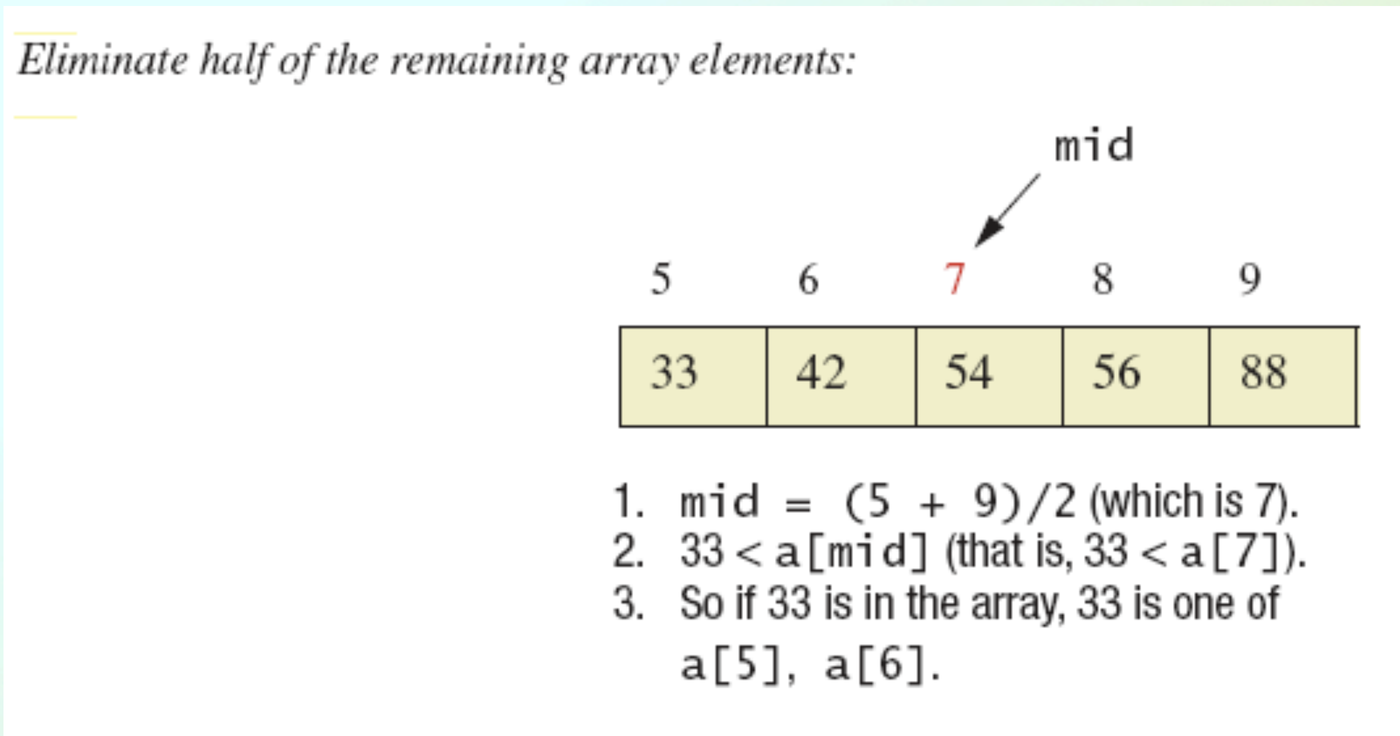
Eliminate half of the array elements:



1. `mid = (0 + 9)/2` (which is 4).
2. 33 > `a[mid]` (that is, 33 > `a[4]`).
3. So if 33 is in the array, 33 is one of `a[5]`, `a[6]`, `a[7]`, `a[8]`, `a[9]`.

Binary Search

- Figure 11.3b Binary search example

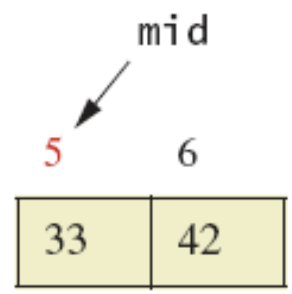


*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

Binary Search

- Figure 11.3c Binary search example

Eliminate half of the remaining array elements:



1. `mid = (5 + 6)/2` (which is 5).
2. 33 equals `a[mid]`, so we found 33 at index `5`.

`33` *found in* `a[5]`.

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch

Binary Search

- View final code, listing 11.6
  `class ArraySearcher`
- Note demo program, listing 11.7
  `class ArraySearcherDemo`

Binary Search

```
Enter 10 integers in increasing order.
Again?
yes
Enter a value to search for:
0
0 is at index 0
Again?
yes
Enter a value to search for:
2
2 is at index 1
Again?
yes
Enter a value to search for:
13
13 is not in the array.
Again?
no
May you find what you're searching for.
```

Sample screen output

Programming Example

- Merge sort – A recursive sorting method
- A divide-and-conquer algorithm
  - Array to be sorted is divided in half
  - The two halves are sorted by recursive calls
  - This produces two smaller, sorted arrays which are merged to a single sorted array

Merge Sort

- Algorithm to sort array **`a`**

1. If the array a has only one element, do nothing (base case).
   Otherwise, do the following (recursive case):
2. Copy the first half of the elements in a to a smaller array named `firstHalf`.
3. Copy the rest of the elements in the array a to another smaller array named `lastHalf`.
4. Sort the array `firstHalf` using a recursive call.
5. Sort the array `lastHalf` using a recursive call.
6. Merge the elements in the arrays `firstHalf` and `lastHalf` into the array a.

- View Java implementation, listing 11.8
  `class MergeSort`

JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709

Merge Sort

- View demo program, listing 11.9

  `class MergeSortDemo`

```
Array values before sorting:
7 5 11 2 16 4 18 14 12 30
Array values after sorting:
2 4 5 7 11 12 14 16 18 30
```

Sample screen output

ISBN 0132162709

Summary

- Method with self invocation
  - Invocation considered a recursive call
- Recursive calls
  - Legal in Java
  - Can make some method definitions clearer
- Algorithm with one subtask that is smaller version of entire task
  - Algorithm is a recursive method

Summary

- To avoid infinite recursion recursive method should contain two kinds of cases
  - A recursive call
  - A base (stopping) case with no recursive call
- Good examples of recursive algorithms
  - Binary search algorithm
  - Merge sort algorithm