Recursion

Chapter 11
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

• 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
Case Study

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

• Heading

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

• Consider this useful private method

```java
// 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 11.1

```java
class RecursionDemo
```

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 11.1a Executing recursive call

```java
// Code for invocation of displayAsWords(987)
if (987 < 10)
    System.out.print(getWordFromDigit(987) + " ");
else // 987 has two or more digits
{
    // Computation waits here for the completion of the recursive call.
    System.out.print(getWordFromDigit(987 % 10) + " ");
}
```

displayAsWords(987) is equivalent to executing:
How Recursion Works

• Figure 11.1b Executing recursive call

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

```java
//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) + " ");
    }
```
How Recursion Works

• Figure 11.1c Executing recursive call

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

```java
//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) + " ");
}
```
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 use "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

```java
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 11.2
  `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 11.3

*class RecursionDemo2*
Recursive Methods that Return a Value

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

Sample screen output:

Enter a nonnegative number:
2008
2008 contains 2 zeros.
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

• View program, listing 11.4
  class CountDown
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

1. \( m = \) an index between 0 and \((a.length - 1)\)
2. if (target == a[m])
   3. return \( m \);
4. else if (target < a[m])
   5. return the result of searching \( a[0] \) through \( a[m - 1] \)
6. else if (target > a[m])
   7. return the result of searching \( a[m + 1] \) through \( a[a.length - 1] \)

- Algorithm requires additional parameters
Binary Search

• Draft 2 of algorithm to search $a[first]$ through $a[last]$

1. mid = approximate midpoint between first and last
2. if (target == a[mid])
   3. return mid
4. else if (target < a[mid])
   5. return the result of searching $a[first]$ through $a[mid - 1]$
6. else if (target > a[mid])
   7. return the result of searching $a[mid + 1]$ through $a[last]$

- What if target is not in the array?
Binary Search

• Final draft of algorithm to search \( a[\text{first}] \) through \( a[\text{last}] \) to find \( \text{target} \)

1. \( \text{mid} = \text{approximate midpoint between first and last} \)
2. \( \text{if} (\text{first} > \text{last}) \)
3. \( \quad \text{return} -1 \)
4. \( \text{else if} (\text{target} == a[\text{mid}]) \)
5. \( \quad \text{return} \text{mid} \)
6. \( \text{else if} (\text{target} < a[\text{mid}]) \)
7. \( \quad \text{return the result of searching} a[\text{first}] \text{ through } a[\text{mid} - 1] \)
8. \( \text{else if} (\text{target} > a[\text{mid}]) \)
9. \( \quad \text{return the result of searching} a[\text{mid} + 1] \text{ through } a[\text{last}] \)
Binary Search

• Figure 11.2a Binary search example

```
target is 33

Eliminate half of the array elements:

mid

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>32</td>
<td>33</td>
<td>42</td>
<td>54</td>
<td>56</td>
<td>88</td>
</tr>
</tbody>
</table>

1. \( \text{mid} = (0 + 9)/2 \) (which is 4).
2. \( 33 > a[\text{mid}] \) (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 11.2b Binary search example

Eliminate half of the remaining array elements:

mid

5 6 7 8 9
33 42 54 56 88

1. mid = (5 + 9)/2 (which is 7).
2. 33 < a[mid] (that is, 33 < a[7]).
3. So if 33 is in the array, 33 is one of a[5], a[6].
Binary Search

• Figure 11.2c Binary search example

Eliminate half of the remaining array elements:

mid

5 6

33 42

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].
Binary Search

• View final code, listing 11.5 class ArraySearcher

• Note demo program, listing 11.6 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.
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 \( \text{firstHalf} \).
3. Copy the rest of the elements in the array \( a \) to another smaller array named \( \text{lastHalf} \).
4. Sort the array \( \text{firstHalf} \) using a recursive call.
5. Sort the array \( \text{lastHalf} \) using a recursive call.
6. Merge the elements in the arrays \( \text{firstHalf} \) and \( \text{lastHalf} \) into the array \( a \).

• View [Java implementation](#), listing 11.7

```java
class MergeSort
```
Merge Sort

• View demo program, listing 11.8
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
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