



# D06

# PROGRAMMING with JAVA

## Ch18 – Recursion

PowerPoint presentation, created by Angel A. Juan - [ajuanp@gmail.com](mailto:ajuanp@gmail.com),  
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

# Chapter Goals

---

- To learn about the **method of recursion**
- To understand the relationship between recursion and **iteration**
- To analyze problems that are much easier to solve by recursion than by iteration
- To learn to "think recursively"
- To be able to use recursive helper methods
- To understand when the use of recursion affects the efficiency of an algorithm

# Triangle Numbers

- Problem of the  $n$ th triangle number: We want to compute the area of a triangle of width  $n$ , assuming that each [ ] square has an area of 1. For example, the third triangle number is 6:

```
[ ]
[ ][ ]      Area = 6
[ ][ ][ ]
```

- Here is the outline of the class that we will develop:

```
public class Triangle
{
    public Triangle(int aWidth)
    {
        width = aWidth;
    }
    public int getArea()
    {
        . . .
    }
    private int width;
}
```

# Handling Triangle of Width 1

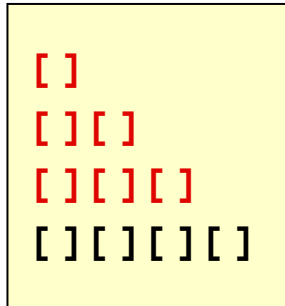
---

- The triangle consists of a single square. Its area is 1
- Add the code to `getArea` method for width 1:

```
public int getArea()  
{  
    if (width == 1) return 1;  
    . . .  
}
```

# Handling the General Case

- Assume we know the area of the smaller, colored triangle



- Area of larger triangle can be calculated as:

```
smallerArea + width
```

- To get the area of the smaller triangle, make a smaller triangle and ask it for its area

```
Triangle smallerTriangle = new Triangle(width - 1);  
int smallerArea = smallerTriangle.getArea();
```

# Completed `getArea` method

```
public int getArea()  
{  
    if (width == 1) return 1;  
    Triangle smallerTriangle = new Triangle(width - 1);  
    int smallerArea = smallerTriangle.getArea();  
    return smallerArea + width;  
}
```

The complete `getArea` method

- `getArea` method makes a smaller triangle of width 3
    - It calls `getArea` on that triangle
      - That method makes a smaller triangle of width 2
        - It calls `getArea` on that triangle
          - That method makes a smaller triangle of width 1
            - It calls `getArea` on that triangle
- The method returns `smallerArea + width = 1 + 2 = 3`
  - The method returns `smallerArea + width = 3 + 3 = 6`
- The method returns `smallerArea + width = 6 + 4 = 10`

Here is an illustration of what happens when we compute the area of a triangle of width 4

To solve the area problem for a triangle of a given width, we use the fact that we can solve the same problem for a lesser width. This is called a **recursive solution**.

# Recursion

---

- A recursive computation solves a problem by using the solution of the same problem with simpler values
- For recursion to terminate, there must be special cases for the simplest inputs.
- To complete our Triangle ex., we must handle width  $\leq 0$ :

```
if (width <= 0) return 0;
```



Two key requirements for recursion success:

- Every recursive call must simplify the computation in some way
- There must be special cases to handle the simplest computations directly

# Other Ways to Compute Triangle Numbers

- Recursion is not really necessary to compute the triangle numbers. There are several ways to solve this simple problem:

- The area of a triangle equals the sum

`1 + 2 + 3 + . . . + width`

- Using a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
    area = area + i;
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions can be complex

- Using math:

```
1 + 2 + . . . + n = n * (n + 1) / 2
=> width * (width + 1) / 2
```

# File Triangle.java

```
01: /**
02:     A triangular shape composed of stacked unit squares like this:
03:     []
04:     [][]
05:     [][][]
06:     . . .
07: */
08: public class Triangle
09: {
10:     /**
11:         Constructs a triangular shape.
12:         @param aWidth the width (and height) of the triangle
13:     */
14:     public Triangle(int aWidth)
15:     {
16:         width = aWidth;
17:     }
```

*Continued*

# File Triangle.java

```
18:
19:     /**
20:         Computes the area of the triangle.
21:         @return the area
22:     */
23:     public int getArea()
24:     {
25:         if (width <= 0) return 0;
26:         if (width == 1) return 1;
27:         Triangle smallerTriangle = new Triangle(width - 1);
28:         int smallerArea = smallerTriangle.getArea();
29:         return smallerArea + width;
30:     }
31:
32:     private int width;
33: }
```

# File TriangleTester.java

```
01: import java.util.Scanner;
02:
03: public class TriangleTester
04: {
05:     public static void main(String[] args)
06:     {
07:         Scanner in = new Scanner(System.in);
08:         System.out.print("Enter width: ");
09:         int width = in.nextInt();
10:         Triangle t = new Triangle(width);
11:         int area = t.getArea();
12:         System.out.println("Area = " + area);
13:     }
14: }
```

## Output

```
Enter width: 10
Area = 55
```

# Self Check

---

1. Why is the statement

```
if (width == 1) return 1;
```

in the `getArea` method unnecessary?

2. How would you modify the program to recursively compute the area of a square?

# Answers

1. Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1, to arrive at the correct area.
2. You would compute the smaller area recursively, then return

```
smallerArea + width + width - 1.
```

```
[ ] [ ] [ ] [ ] [ ]  
[ ] [ ] [ ] [ ] [ ]  
[ ] [ ] [ ] [ ] [ ]  
[ ] [ ] [ ] [ ] [ ]
```

Of course, it would be simpler to compute:

$$1 + 0 + 2 + 1 + 3 + 2 + \dots + n + n - 1 = \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2.$$

# Permutations

- A more difficult example of recursion: we want to design a class that lists all permutations of a string (ex.: the string “eat” has six permutations: “eat”, “eta”, “aet”, “tea”, and “tae”)
- We will define a class that is in charge of computing the answer. In this case the answer is not a single number but collection of permuted strings:

```
public class PermutationGenerator
{
    public PermutationGenerator(String aWord) { . . . }
    ArrayList<String> getPermutations() { . . . }
}
```

# To Generate All Permutations

---

- A way to generate the permutations recursively: First, we'll generate all permutations that start with 'e', then those that start with 'a', and finally those that start with 't'; to generate permutations starting with 'e', we need to find all permutations of "at" (same problem with simpler inputs → use recursion!)
- Starting with 'e' → generate the permutations of the substring "at" ("at", "ta"); for each permutation of that substring, prepend the letter 'e' to get the permutations of "eat" that start with 'e' ("eat", "eta")
- Starting with 'a' → generate the permutations of the substring "et" ("et", "te"); for each permutation of that substring, prepend the letter 'a' to get the permutations of "eat" that start with 'a' ("aet", "ate")
- Starting with 't' → use the same methodology...

*Continued*

# To Generate All Permutations

- In the `getPermutations` method, we loop through all positions in the word to be permuted. For each of them, we compute the shorter word that is obtained by removing the `i`th letter:

```
String shorterWord = word.substring(0, i) + word.substring(i + 1);
```

- We construct a permutation generator to get the permutations of the shorter word, and ask it to give us all permutations of the shorter word:

```
PermutationGenerator shorterPermutationGenerator  
    = new PermutationGenerator(shorterWord);  
ArrayList<String> shorterWordPermutations  
    = shorterPermutationGenerator.getPermutations();
```

*Continued*

# To Generate All Permutations

---

- Finally, add the removed letter to the front of all permutations of the shorter word:

```
for (String s : shorterWordPermutations)
{
    result.add(word.charAt(i) + s);
}
```

- Special case: simplest possible string is the empty string, which has a single permutation – itself

# File PermutationGenerator.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     This class generates permutations of a word.
05: */
06: public class PermutationGenerator
07: {
08:     /**
09:         Constructs a permutation generator.
10:         @param aWord the word to permute
11:     */
12:     public PermutationGenerator(String aWord)
13:     {
14:         word = aWord;
15:     }
16:
```

*Continued*

# File PermutationGenerator.java

```
17:    /**
18:        Gets all permutations of a given word.
19:    */
20:    public ArrayList<String> getPermutations()
21:    {
22:        ArrayList<String> result = new ArrayList<String>();
23:
24:        // The empty string has a single permutation: itself
25:        if (word.length() == 0)
26:        {
27:            result.add(word);
28:            return result;
29:        }
30:
31:        // Loop through all character positions
32:        for (int i = 0; i < word.length(); i++)
33:        {
```

*Continued*

# File PermutationGenerator.java

```
34:         // Form a simpler word by removing the ith character
35:         String shorterWord = word.substring(0, i)
36:             + word.substring(i + 1);
37:
38:         // Generate all permutations of the simpler word
39:         PermutationGenerator shorterPermutationGenerator
40:             = new PermutationGenerator(shorterWord);
41:         ArrayList<String> shorterWordPermutations
42:             = shorterPermutationGenerator.getPermutations();
43:
44:         // Add the removed character to the front of
45:         // each permutation of the simpler word,
46:         for (String s : shorterWordPermutations)
47:         {
48:             result.add(word.charAt(i) + s);
49:         }
50:     }
```

*Continued*

# File PermutationGenerator.java

---

```
51:         // Return all permutations
52:         return result;
53:     }
54:
55:     private String word;
56: }
```

# File

## PermutationGeneratorTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     This program tests the permutation generator.
05: */
06: public class PermutationGeneratorTester
07: {
08:     public static void main(String[] args)
09:     {
10:         PermutationGenerator generator
11:             = new PermutationGenerator("eat");
12:         ArrayList<String> permutations
13:             = generator.getPermutations();
14:         for (String s : permutations)
15:         {
16:             System.out.println(s);
17:         }
18:     }
19: }
```

*Continued*

# File

## PermutationGeneratorTester.java

---

```
17:     }  
18: }  
19:
```

## Output

```
eat  
eta  
aet  
ate  
tea  
tae
```

# Self Check

---

3. What are all permutations of the four-letter word `beat`?
4. Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?

# Answers

---

3. They are **b** followed by the six permutations of **eat**, **e** followed by the six permutations of **bat**, **a** followed by the six permutations of **bet**, and **t** followed by the six permutations of **bea**.
4. Simply change

```
if (word.length() == 0)  if (word.length() <= 1)
```

because a word with a single letter is also its sole permutation.

# Thinking Recursively

---



If you need to solve a complex problem, pretend that “someone else” will do most of the heavy lifting and solve the problem for all simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem

- Problem: test whether a sentence is a **palindrome** (a string that is equal to itself when you reverse all characters, like “Madam, I’m Adam”)

# Implement isPalindrome Method

```
public class Sentence
{
    /**
     Constructs a sentence.
     @param aText a string containing all characters of
     the sentence
    */
    public Sentence(String aText)
    {
        text = aText;
    }

    /**
     Tests whether this sentence is a palindrome.
     @return true if this sentence is a palindrome,
     false otherwise
    */
    public boolean isPalindrome()
    {
        . . .
    }
    private String text;
}
```

# Thinking Recursively: Step-by-Step

---

1. Consider various ways to simplify inputs  
Here are several possibilities:
  - Remove the first character
  - Remove the last character
  - Remove both the first and last characters
  - Remove a character from the middle

*Continued*

# Thinking Recursively: Step-by-Step

---

2. Combine solutions with simpler inputs into a solution of the original problem
  - Most promising simplification: remove first and last characters  
"adam, I'm Ada", is a palindrome too!
  - Thus, a word is a palindrome if
    - The first and last letters match, and
    - Word obtained by removing the first and last letters is a palindrome

*Continued*

# Thinking Recursively: Step-by-Step

---

2. Combine solutions with simpler inputs into a solution of the original problem
  - What if first or last character is not a letter? Ignore it
    - If the first and last characters are letters, check whether they match;  
if so, remove both and test shorter string
    - If last character isn't a letter, remove it and test shorter string
    - If first character isn't a letter, remove it and test shorter string

*Continued*

# Thinking Recursively: Step-by-Step

---

## 3. Find solutions to the simplest inputs

- Strings with two characters
  - No special case required; step two still applies
- Strings with a single character
  - They are palindromes
- The empty string
  - It is a palindrome

*Continued*

# Thinking Recursively: Step-by-Step

4. Implement the solution by combining the simple cases and the reduction step

```
public boolean isPalindrome()  
{  
    int length = text.length();  
  
    // Separate case for shortest strings.  
    if (length <= 1) return true;  
  
    // Get first and last characters, converted to lowercase.  
    char first = Character.toLowerCase(text.charAt(0));  
    char last = Character.toLowerCase(text.charAt(length - 1));
```

*Continued*

# Thinking Recursively: Step-by-Step

```
if (Character.isLetter(first) && Character.isLetter(last))
{
    // Both are letters.
    if (first == last)
    {
        // Remove both first and last character.
        Sentence shorter
            = new Sentence(text.substring(1, length - 1));
        return shorter.isPalindrome();
    }
    else
        return false;
}
```

*Continued*

# Thinking Recursively: Step-by-Step

```
else if (!Character.isLetter(last))
{
    // Remove last character.
    Sentence shorter
        = new Sentence(text.substring(0, length - 1));
    return shorter.isPalindrome();
}
else
{
    // Remove first character.
    Sentence shorter = new Sentence(text.substring(1));
    return shorter.isPalindrome();
}
}
```

# Recursive Helper Methods

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem
- Consider the palindrome test of previous slide  
It is a bit inefficient to construct new `Sentence` objects in every step
- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
/**
 * Tests whether a substring of the sentence is a palindrome.
 * @param start the index of the first character of the
 *           substring
 * @param end the index of the last character of the substring
 * @return true if the substring is a palindrome
 */
public boolean isPalindrome(int start, int end)
```

- Then, simply call the helper method with positions that test the entire string:

```
public boolean isPalindrome()
{
    return isPalindrome(0, text.length() - 1);
}
```

# Recursive Helper Methods: isPalindrome

```
public boolean isPalindrome(int start, int end)
{
    // Separate case for substrings of length 0 and 1.
    if (start >= end) return true;

    // Get first and last characters, converted to lowercase.
    char first = Character.toLowerCase(text.charAt(start));
    char last = Character.toLowerCase(text.charAt(end));
```

*Continued*

# Recursive Helper Methods: isPalindrome

```
if (Character.isLetter(first) && Character.isLetter(last))
{
    if (first == last)
    {
        // Test substring that doesn't contain the
        // matching letters.
        return isPalindrome(start + 1, end - 1);
    }
    else
        return false;
}
else if (!Character.isLetter(last))
{
    // Test substring that doesn't contain the last character.
    return isPalindrome(start, end - 1);
}
```

*Continued*

# Recursive Helper Methods: isPalindrome

---

```
else
{
    // Test substring that doesn't contain the first character.
    return isPalindrome(start + 1, end);
}
}
```

# Self Check

---

5. Do we have to give the same name to both `isPalindrome` methods?
6. When does the recursive `isPalindrome` method stop calling itself?

# Answers

---

5. No—the first one could be given a different name such as `substringIsPalindrome`.
6. When `start >= end`, that is, when the investigated string is either empty or has length 1.

# Fibonacci Sequence

- **Fibonacci sequence** is a sequence of numbers defined by:

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

- **First ten terms:**

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

# File FibTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program computes Fibonacci numbers using a recursive
05:     method.
06: */
07: public class FibTester
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         System.out.print("Enter n: ");
13:         int n = in.nextInt();
14:
15:         for (int i = 1; i <= n; i++)
16:         {
```

*Continued*

# File FibTester.java

```
17:         long f = fib(i);
18:         System.out.println("fib(" + i + ") = " + f);
19:     }
20: }
21:
22: /**
23:     Computes a Fibonacci number.
24:     @param n an integer
25:     @return the nth Fibonacci number
26: */
27: public static long fib(int n)
28: {
29:     if (n <= 2) return 1;
30:     else return fib(n - 1) + fib(n - 2);
31: }
32: }
```

*Continued*

# File FibTester.java

---

## Output

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

# The Efficiency of Recursion

---

- Recursive implementation of `fib` is straightforward
- Watch the output closely as you run the test program
- First few calls to `fib` are quite fast
- For larger values, the program pauses an amazingly long time between outputs
- To find out the problem, lets insert trace messages

# File FibTrace.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program prints trace messages that show how often the
05:     recursive method for computing Fibonacci numbers calls itself.
06: */
07: public class FibTrace
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         System.out.print("Enter n: ");
13:         int n = in.nextInt();
14:
15:         long f = fib(n);
16:
17:         System.out.println("fib(" + n + ") = " + f);
18:     }
```

*Continued*

# File FibTrace.java

```
19:
20:     /**
21:         Computes a Fibonacci number.
22:         @param n an integer
23:         @return the nth Fibonacci number
24:     */
25:     public static long fib(int n)
26:     {
27:         System.out.println("Entering fib: n = " + n);
28:         long f;
29:         if (n <= 2) f = 1;
30:         else f = fib(n - 1) + fib(n - 2);
31:         System.out.println("Exiting fib: n = " + n
32:             + " return value = " + f);
33:         return f;
34:     }
35: }
```

# File FibTrace.java

## Output

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
```

```
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

# Call Tree for Computing fib(6)

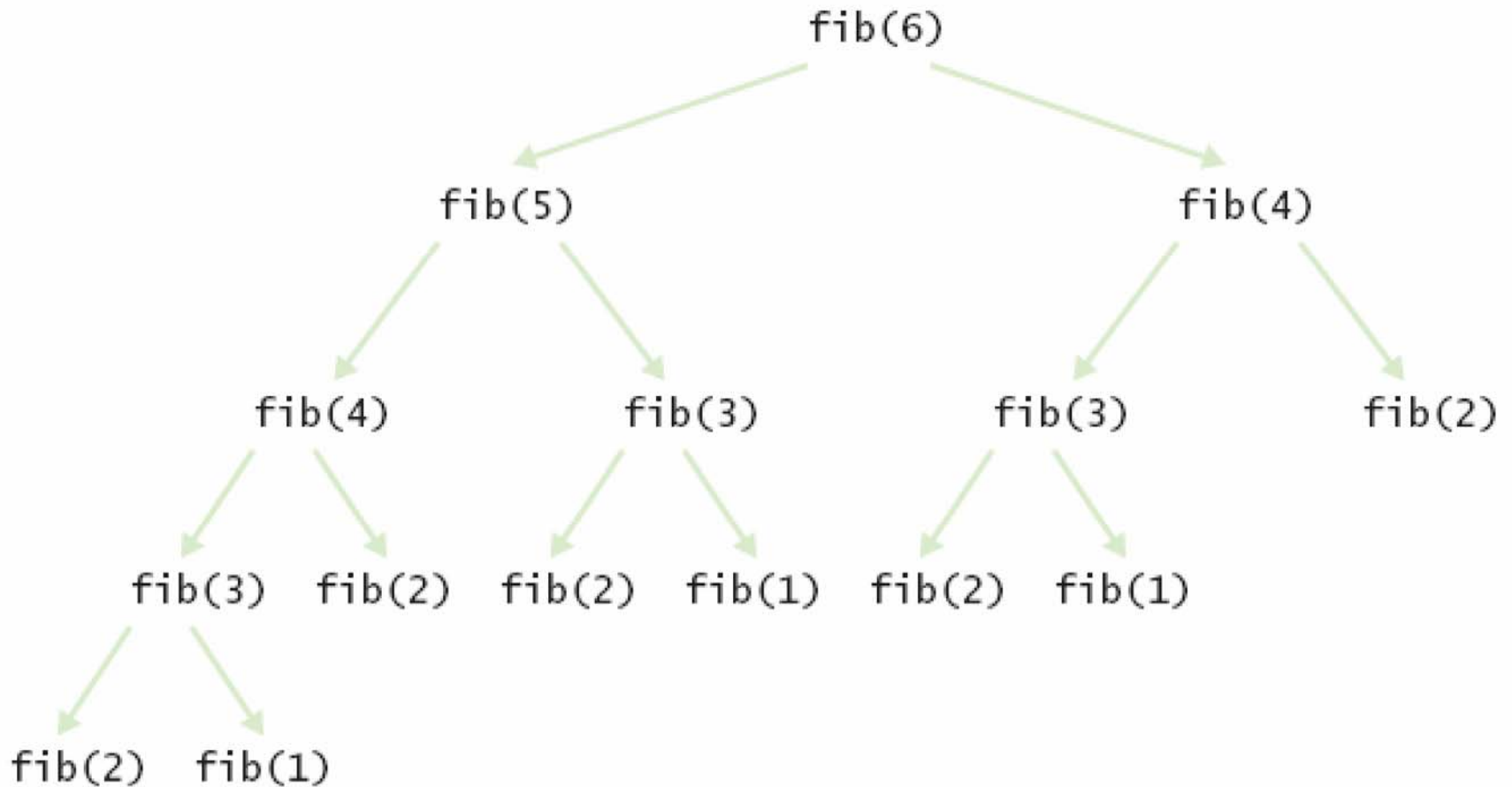


Figure 2:  
Call Tree of the Recursive fib method

# The Efficiency of Recursion

---

- Method takes so long because it computes the same values over and over
- The computation of `fib(6)` calls `fib(3)` three times
- Imitate the pencil-and-paper process to avoid computing the values more than once

# File FibLoop.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program computes Fibonacci numbers using an
        // iterative method.
05: */
06: public class FibLoop
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:         System.out.print("Enter n: ");
12:         int n = in.nextInt();
13:
14:         for (int i = 1; i <= n; i++)
15:         {
```

*Continued*

# File FibLoop.java

```
16:         long f = fib(i);
17:         System.out.println("fib(" + i + ") = " + f);
18:     }
19: }
20:
21: /**
22:     Computes a Fibonacci number.
23:     @param n an integer
24:     @return the nth Fibonacci number
25: */
26: public static long fib(int n)
27: {
28:     if (n <= 2) return 1;
29:     long fold = 1;
30:     long fold2 = 1;
31:     long fnew = 1;
32:     for (int i = 3; i <= n; i++)
33:     {
```

*Continued*

# File FibLoop.java

```
34:         fnew = fold + fold2;
35:         fold2 = fold;
36:         fold = fnew;
37:     }
38:     return fnew;
39: }
40: }
```

## Output

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

# The Efficiency of Recursion

---

- Occasionally, a recursive solution runs much slower than its iterative counterpart
- In most cases, the recursive solution is only slightly slower
- The iterative `isPalindrome` performs only slightly better than recursive solution
  - Each recursive method call takes a certain amount of processor time
- Smart compilers can avoid recursive method calls if they follow simple patterns. Most compilers don't do that
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution
- "To iterate is human, to recurse divine.", L. Peter Deutsch

# Iterative isPalindrome Method

```
public boolean isPalindrome()
{
    int start = 0;
    int end = text.length() - 1; while (start < end)
    {
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));
        if (Character.isLetter(first) && Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {
                start++;
                end--;
            }
            else
                return false;
        }
        if (!Character.isLetter(last))
            end--;
        if (!Character.isLetter(first))
            start++;
    }
    return true;
}
```

# Self Check

---

7. You can compute the factorial function either with a loop, using the definition that  $n! = 1 \times 2 \times \dots \times n$ , or recursively, using the definition that  $0! = 1$  and  $n! = (n - 1)! \times n$ . Is the recursive approach inefficient in this case?
8. Why isn't it easy to develop an iterative solution for the permutation generator?

# Answers

---

7. No, the recursive solution is about as efficient as the iterative approach. Both require  $n - 1$  multiplications to compute  $n!$ .
8. An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious

# Chapter Summary

---

- A **recursive computation** solves a problem by using the solution of the same problem with simpler values
- For a recursion to terminate, there must be special cases for the simplest values
- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem
- Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution