34. Recursion

Java

Summer 2008

Instructor: Dr. Masoud Yaghini

Outline

- Introduction
- Example: Factorials
- Example: Fibonacci Numbers
- Recursion vs. Iteration
- References

Introduction

Introduction

- **Recursive methods**
  - A method that invokes itself directly or indirectly.
- Recursion is a useful programming technique.
- In some cases, using recursion enables you to develop a natural, straightforward, simple solution to a problem that would otherwise be difficult to solve.
- Many mathematical functions are defined using recursion.

Example: Factorials

Example: Factorial

- Consider the factorial of a positive integer n, written n! (and pronounced "n factorial"), which is the product

  n x (n - 1) x (n - 2) x … x 1
- with 1! equal to 1 and 0! defined to be 1. For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120.
- The factorial of integer n (where n >= 0) can be calculated iteratively (non-recursively) using a for statement as follows:

```
factorial = 1;
for ( int counter = n; counter >= 1; counter-- )
    factorial = factorial * counter;
```

Recursion

```java
package chapter19;

import javax.swing.JOptionPane;

public class ComputeFactorialIteratively {
   /** Main method */
   public static void main(String[] args) {
      // Prompt the user to enter an integer
      String intString = JOptionPane.showInputDialog(
            "Please enter a non-negative integer:");

      // Convert string into integer
      int n = Integer.parseInt(intString);

      // Display factorial
      JOptionPane.showMessageDialog(null,
            "Factorial of " + n + " is " + factorial(n));
   }

   /** Return the factorial for a specified index */
   static long factorial(int number) {
      long factorial = 1;
      for ( int counter = number; counter >= 1; counter-- )
         factorial = factorial * counter;
      return factorial;
   }
}
```

Example: Factorials

- The factorial of a number n can be recursively defined as follows:
- The factorial of a number n can be recursively defined as follows:
  - $0! = 1$;
  - $n! = n \times (n - 1)!$; $n > 0$

Example: Factorials

- Let factorial(n) be the method for computing n!.
- If you call the method with n = 0, it immediately returns the result.
- The method knows how to solve the simplest case, which is referred to as the ***base case*** or the ***stopping condition***.
- If you call the method with n > 0, it reduces the problem into a subproblem for computing the factorial of n - 1.

Example: Factorials

- The subproblem is essentially the same as the original problem, but is simpler or smaller than the original.
- Because the subproblem has the same property as the original, you can call the method with a different argument, which is referred to as a recursive call.
- The recursive algorithm for computing factorial(n) can be simply described as follows:

```
if (n == 0)
    return 1;
else
    return n * factorial(n - 1);
```

Recursion

```java
package chapter19;

import javax.swing.JOptionPane;

public class ComputeFactorialRecursively {
   /** Main method */
   public static void main(String[] args) {
      // Prompt the user to enter an integer
      String intString = JOptionPane.showInputDialog(
            "Please enter a non-negative integer:");

      // Convert string into integer
      int n = Integer.parseInt(intString);

      // Display factorial
      JOptionPane.showMessageDialog(null,
            "Factorial of " + n + " is " + factorial(n));
   }

   /** Return the factorial for a specified index */
   static long factorial(int number) {
      if (number == 0) // Stopping condition
         return 1;
      else
         return number * factorial(number - 1); // Call factorial recursively
   }
}
```

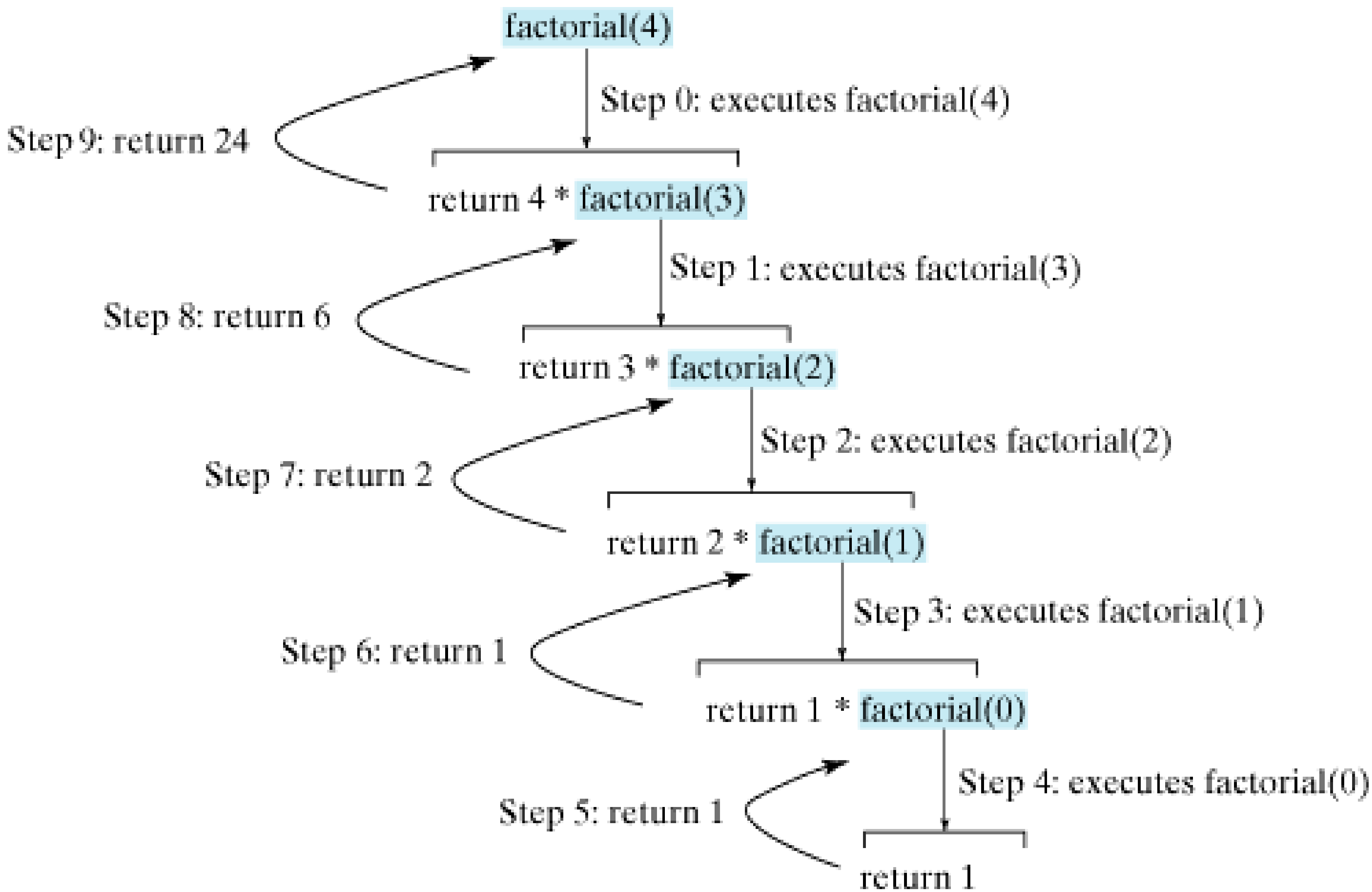
Example: Factorials

- For a recursive method to terminate, the problem must eventually be reduced to a stopping case.
- When it reaches a stopping case, the method returns a result to its caller.
- The caller then performs a computation and returns the result to its own caller.
- This process continues until the result is passed back to the original caller.

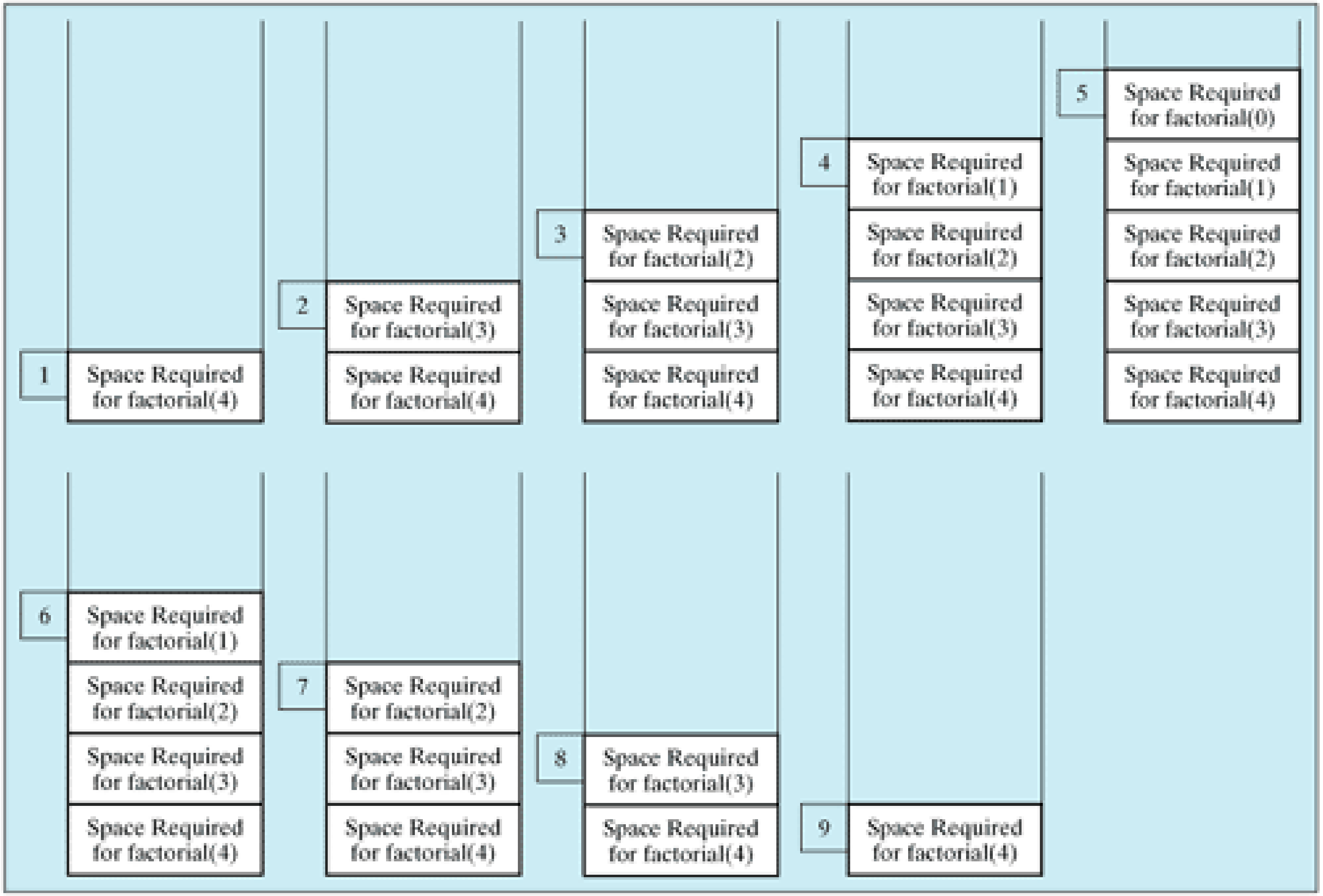
Example: Factorials - Invoking factorial(4)

Factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1))))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24

Example: Factorials - Invoking factorial(4)



Example: Factorials – Memory Space



Caution

- It is simpler and more efficient to implement the factorial method using a loop.
- However, the recursive factorial method is a good example to demonstrate the concept of recursion.

Caution

- ***Infinite recursion*** can occur if recursion does not reduce the problem in a manner that allows it to eventually converge into the base case.
- For example, if you mistakenly write the factorial method as follows:

```
public static long factorial(int n) {
    return n * factorial(n - 1);
}
```

- The method runs infinitely and causes a StackOverflowError.

Example: Fibonacci Numbers

Example: Fibonacci Numbers

- Consider the well-known Fibonacci series problem, as follows:

The series:	0	1	1	2	3	5	8	13	21	34	55	89 …
indices:	0	1	2	3	4	5	6	7	8	9	10	11

- The Fibonacci series begins with 0 and 1, and each subsequent number is the sum of the preceding two numbers in the series.
- The series can be recursively defined as follows:

  fib(0) = 0;

  fib(1) = 1;

  fib(index) = fib(index - 2) + fib(index - 1); index >= 2

Example: Fibonacci Numbers

- The recursive algorithm for computing fib(index) can be simply described as follows:

```
if (index == 0)
    return 0;
else if (index == 1)
    return 1;
else
    return fib(index - 1) + fib(index - 2);
```

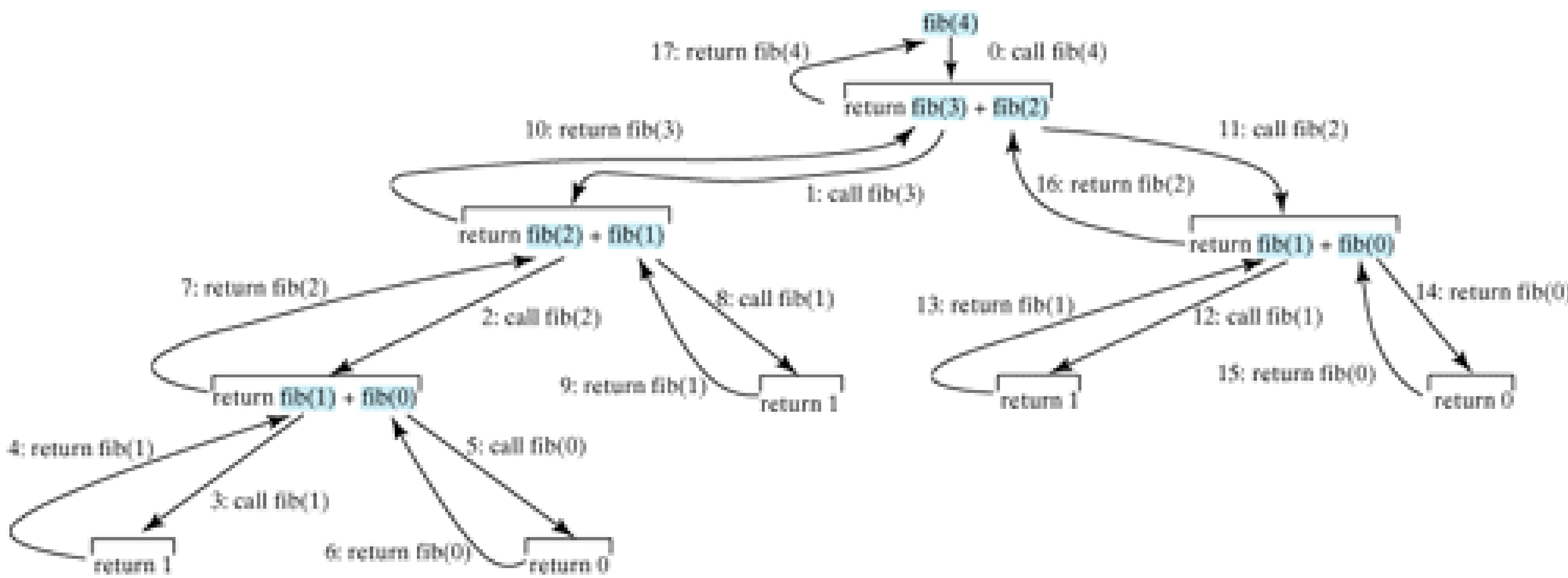
- Example:

```
fib(3)   = fib(2) + fib(1)
         = (fib(1) + fib(0)) + fib(1)
         = (1 + 0) + fib(1)
         = 1 + fib(1)
         = 1 + 1
         = 2
```

Recursion

```java
package chapter19;

import javax.swing.JOptionPane;

public class ComputeFibonacciRecursively {
  /** Main method */
  public static void main(String args[]) {
    // Read the index
    String intString = JOptionPane.showInputDialog(
        "Enter an index for the Fibonacci number:");

    // Convert string into integer
    int index = Integer.parseInt(intString);

    // Find and display the Fibonacci number
    JOptionPane.showMessageDialog(null,
        "Fibonacci number at index " + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(long index) {
    if (index == 0) // Stopping condition
      return 0;
    else if (index == 1) // Stopping condition
      return 1;
    else  // Reduction and recursive calls
      return fib(index - 1) + fib(index - 2);
  }
}
```

Example: Fibonacci Numbers



Example: Fibonacci Numbers

- The recursive implementation of the fib method is very simple and straightforward, but not efficient.
- The recursive fib method is a good example to demonstrate how to write recursive methods, though it is not practical.
- See ComputeFibonacciIteratively.java an efficient solution using loops.

Recursion

```java
package chapter19;

import javax.swing.JOptionPane;

public class ComputeFibonacciIteratively {
  /** Main method */
  public static void main(String args[]) {
    // Read the index
    String intString = JOptionPane.showInputDialog(
        "Enter an index for the Fibonacci number:");

    // Convert string into integer
    int index = Integer.parseInt(intString);

    // Find and display the Fibonacci number
    JOptionPane.showMessageDialog(null,
        "Fibonacci number at index (Iteratively) " + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(int n) {
    int f0 = 0, f1 = 1, currentFib;

    if (n == 0) return 0;
    if (n == 1) return 1;

    for (int i = 1; i < n; i++) {
      currentFib = f0+f1;
      f0 = f1;
      f1 = currentFib;
    }
    return f1;
  }
}
```

Recursion vs. Iteration

Recursion vs. Iteration

- All recursive methods have the following characteristics:
  - The method is implemented using an if-else or a switch statement that leads to different cases.
  - One or more base cases (the simplest case) are used to stop recursion.
  - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Recursion vs. Iteration

- In general, to solve a problem using recursion, you break it into subproblems.
- If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively.
- This subproblem is almost the same as the original problem in nature with a smaller size.

Recursion vs. Iteration

- Both iteration and recursion use a control statement
  - Iteration uses a repetition statement,
    - e.g., for, while or do...while
  - Recursion uses a selection statement
    - e.g., if, if...else or switch

Recursion vs. Iteration

- Both iteration and recursion involve repetition:
  - Iteration explicitly uses a repetition statement,
  - Recursion achieves repetition through repeated method calls.
- Iteration and recursion both involve a termination test
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a base case is reached

Recursion vs. Iteration

- A recursive approach is normally preferred over an iterative approach when:
  - The recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug.
  - A recursive approach can often be implemented with fewer lines of code.

Recursion vs. Iteration

- Any problem that can be solved recursively can also be solved iteratively.
- Recursion can be expensive in terms of processor time and memory space
- Avoid using recursion in situations requiring high performance. Recursive calls take time and consume additional memory.

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 19)
- H. M. Deitel and P. J. Deitel, **Java™ How to Program**, Sixth Edition, Prentice Hall, 2005. (Chapter 15)

The End