Chapter 5

Recursion
Objectives

Discuss the following topics:
• Recursive Definitions
• Method Calls and Recursion Implementation
• Anatomy of a Recursive Call
• Tail Recursion
• Nontail Recursion
• Indirect Recursion
Objectives (continued)

Discuss the following topics:

- Nested Recursion
- Excessive Recursion
- Backtracking
- Case Study: A Recursive Descent Interpreter
Recursive Definitions

• **Recursive definitions** are programming concepts that define themselves

• A recursive definition consists of two parts:
  – The **anchor** or **ground case**, the basic elements that are the building blocks of all other elements of the set
  – Rules that allow for the construction of new objects out of basic elements or objects that have already been constructed
Recursive Definitions (continued)

• Recursive definitions serve two purposes:
  – Generating new elements
  – Testing whether an element belongs to a set

• Recursive definitions are frequently used to define functions and sequences of numbers
Method Calls and Recursion Implementation

• Activation records contain the following:
  – Values for all parameters to the method, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items
  – Local (automatic) variables that can be stored elsewhere
  – The return address to resume control by the caller, the address of the caller’s instruction immediately following the call
Method Calls and Recursion Implementation (continued)

- A dynamic link, which is a pointer to the caller’s activation record
- The returned value for a method not declared as void
Method Calls and Recursion Implementation (continued)

Figure 5-1 Contents of the run-time stack when `main()` calls method `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`
## Anatomy of a Recursive Call

### Figure 5-2 Changes to the run-time stack during execution of `power(5.6, 2)`

<table>
<thead>
<tr>
<th></th>
<th><code>AR for main()</code></th>
<th><code>First call to </code></th>
<th><code>Second call to </code></th>
<th><code>Third call to</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
</tr>
<tr>
<td><code>y</code></td>
<td></td>
<td><code>2 ← SP</code></td>
<td><code>1 ← SP</code></td>
<td><code>0 ← SP</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(136)</td>
<td>(105)</td>
<td>(105)</td>
</tr>
<tr>
<td><code>?</code></td>
<td><code>y</code></td>
<td><code>2 ← SP</code></td>
<td><code>1 ← SP</code></td>
<td><code>0 ← SP</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(136)</td>
<td>(105)</td>
<td>(105)</td>
</tr>
<tr>
<td><code>?</code></td>
<td><code>y</code></td>
<td><code>2 ← SP</code></td>
<td><code>1 ← SP</code></td>
<td><code>0 ← SP</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(136)</td>
<td>(105)</td>
<td>(105)</td>
</tr>
<tr>
<td><code>?</code></td>
<td><code>y</code></td>
<td><code>2 ← SP</code></td>
<td><code>1 ← SP</code></td>
<td><code>0 ← SP</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(136)</td>
<td>(105)</td>
<td>(105)</td>
</tr>
<tr>
<td><code>?</code></td>
<td><code>y</code></td>
<td><code>2 ← SP</code></td>
<td><code>1 ← SP</code></td>
<td><code>0 ← SP</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
<td><code>5.6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(136)</td>
<td>(105)</td>
<td>(105)</td>
</tr>
</tbody>
</table>

**Key:**
- **SP:** Stack pointer
- **AR:** Activation record
- **?** Location reserved for returned value.

Data Structures and Algorithms in Java
Tail Recursion

- **Tail recursion** is characterized by the use of only one recursive call at the very end of a method implementation

```java
void tail (int i) {
    if (i > 0) {
        System.out.print (i + "");
        tail(i-1);
    }
}
```
Tail Recursion

Example of nontail recursion:

```java
void nonTail (int i) {
    if (i > 0) {
        nonTail(i-1);
        System.out.print (i + "");
        nonTail(i-1);
    }
}
```
Nontail Recursion (continued)

Figure 5-3 Changes on the run-time stack during the execution of `reverse()`
Nontail Recursion (continued)

1. Divide an interval \( \text{side} \) into three even parts
2. Move one-third of \( \text{side} \) in the direction specified by \( \text{angle} \)

Figure 5-4 Examples of von Koch snowflakes
Nontail Recursion (continued)

3. Turn to the right 60° (i.e., turn –60°) and go forward one-third of side
4. Turn to the left 120° and proceed forward one-third of side
5. Turn right 60° and again draw a line one-third of side long
Nontail Recursion (continued)

Figure 5-5 The process of drawing four sides of one segment of the von Koch snowflake
Nontail Recursion (continued)

drawFourLines (side, level)
    if (level = 0)
        draw a line;
    else
        drawFourLines(side/3, level-1);
        turn left 60°;
        drawFourLines(side/3, level-1);
        turn right 120°;
        drawFourLines(side/3, level-1);
        turn left 60°;
        drawFourLines(side/3, level-1);
Nontail Recursion (continued)

```java
import java.awt.*;
import java.awt.event.*;

public class vonKoch extends Frame implements ActionListener {
    private TextField lvl, len;
    vonKoch() {
        super("von Koch snowflake");
        Label lvlLbl = new Label("level");
        lvl = new TextField("4",3);
        Label lenLbl = new Label("side");
        len = new TextField("200",3);
        Button draw = new Button("draw");
        lvl.addActionListener(this);
        len.addActionListener(this);
        draw.addActionListener(this);
        setLayout(new FlowLayout());
        add(lvlLbl);
        add(lvl);
        add(lenLbl);
        add(len);
        add(draw);
        setSize(600,400);
        setForeground(Color.white);
    }
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake
Nontail Recursion (continued)

```java
setBackground(Color.red);
show();
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

private double angle;
private Point currPt, pt = new Point();
private void right(double x) {
    angle += x;
}
private void left (double x) {
    angle -= x;
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)
Nontail Recursion (continued)

```java
private void drawFourLines(double side, int level, Graphics g) {
    if (level == 0) {
        // arguments to sin() and cos() must be angles given in radians,
        // thus, the angles given in degrees must be multiplied by PI/180;
        PI/180;
        pt.x = ((int)(Math.cos(angle*Math.PI/180)*side)) + currPt.x;
        pt.y = ((int)(Math.sin(angle*Math.PI/180)*side)) + currPt.y;
        g.drawLine(currPt.x, currPt.y, pt.x, pt.y);
        currPt.x = pt.x;
        currPt.y = pt.y;
    }
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)
Nontail Recursion (continued)

```java
} else {
    drawFourLines(side/3.0, level-1, g);
    left(60);
    drawFourLines(side/3.0, level-1, g);
    right(120);
    drawFourLines(side/3.0, level-1, g);
    left(60);
    drawFourLines(side/3.0, level-1, g);
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)
Nontail Recursion (continued)

```java
public void paint(Graphics g) {
    int level = Integer.parseInt(lvl.getText().trim());
    double side = Double.parseDouble(len.getText().trim());
    currPt = new Point(200,150);
    angle = 0;
    for (int i = 1; i <= 3; i++) {
        drawFourLines(side,level,g);
        right(120);
    }
}

static public void main(String[] a) {
    new vonKoch();
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)
Indirect Recursion

receive(buffer)
    while buffer is not filled up
        if information is still incoming
            get a character and store it in buffer;
        else exit();
        decode(buffer);

decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    transfer information from buffer to file;
    receive(buffer);
Indirect Recursion (continued)

![Diagram of recursive calls for \( \sin(x) \)]

**Figure 5-7** A tree of recursive calls for \( \sin(x) \)
Excessive Recursion

Figure 5-8 The tree of calls for $\text{Fib}(6)$
Excessive Recursion (continued)

<table>
<thead>
<tr>
<th>n</th>
<th>Fib(n+1)</th>
<th>Number of Additions</th>
<th>Number of Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>13</td>
<td>12</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>89</td>
<td>88</td>
<td>177</td>
</tr>
<tr>
<td>15</td>
<td>987</td>
<td>986</td>
<td>1,973</td>
</tr>
<tr>
<td>20</td>
<td>10,946</td>
<td>10,945</td>
<td>21,891</td>
</tr>
<tr>
<td>25</td>
<td>121,393</td>
<td>121,392</td>
<td>242,785</td>
</tr>
<tr>
<td>30</td>
<td>1,346,269</td>
<td>1,346,268</td>
<td>2,692,537</td>
</tr>
</tbody>
</table>

Figure 5-9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers
Excessive Recursion (continued)

![Comparison of iterative and recursive algorithms for calculating Fibonacci numbers](image)

<table>
<thead>
<tr>
<th>n</th>
<th>Number of Additions</th>
<th>Assignments</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Iterative Algorithm</td>
<td>Recursive Algorithm</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>27</td>
<td>177</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>42</td>
<td>1,973</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>57</td>
<td>21,891</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>72</td>
<td>242,785</td>
</tr>
<tr>
<td>30</td>
<td>29</td>
<td>87</td>
<td>2,692,537</td>
</tr>
</tbody>
</table>

Figure 5-10 Comparison of iterative and recursive algorithms for calculating Fibonacci numbers
Backtracking

- **Backtracking** is a technique for returning to a given position (e.g., entry point) after trying other avenues that are unsuccessful in solving a particular problem
Backtracking (continued)

Figure 5-11 The eight queens problem
Backtracking (continued)

putQueen(row)

for every position \( \text{col} \) on the same \( \text{row} \)

if position \( \text{col} \) is available

place the next queen in position \( \text{col} \);

if (row < 8)

putQueen(row+1);

else success;

remove the queen from position \( \text{col} \);
Backtracking (continued)

![Figure 5-12  A 4 x 4 chessboard](image)

Figure 5-12  A 4 x 4 chessboard
import java.io.*;

class Queens {
    final boolean available = true;
    final int squares = 4, norm = squares - 1;
    int[] positionInRow = new int[squares];
    boolean[] column = new boolean[squares];
    boolean[] leftDiagonal = new boolean[squares*2 - 1];
    boolean[] rightDiagonal = new boolean[squares*2 - 1];
    int howMany = 0;
    Queens() {
        for (int i = 0; i < squares; i++) {
            positionInRow[i] = -1;
            column[i] = available;
        }
        for (int i = 0; i < squares*2 - 1; i++)
            leftDiagonal[i] = rightDiagonal[i] = available;
    }
}

Figure 5-13 Eight queens problem implementation
void PrintBoard(PrintStream out) {
    .......
}
void PutQueen(int row) {
    for (int col = 0; col < squares; col++)
        if (column[col] == available &&
            leftDiagonal[row+col] == available &&
            rightDiagonal[row-col+norm] == available) {
            positionInRow[row] = col;
            column[col] = !available;
            leftDiagonal[row+col] = !available;
            rightDiagonal[row-col+norm] = !available;
            if (row < squares-1)
                PutQueen(row+1);
            else PrintBoard(System.out);
            column[col] = available;
            leftDiagonal[row+col] = available;
            rightDiagonal[row-col+norm] = available;
        }
}

Figure 5-13 Eight queens problem implementation (continued)
Backtracking (continued)

```java
} static public void main(String args[]) {
    Queens queens = new Queens();
    queens.PutQueen(0);
    System.out.println(queens.howMany + " solutions found.");
}
```

Figure 5-13 Eight queens problem implementation (continued)
Backtracking (continued)

<table>
<thead>
<tr>
<th>Move</th>
<th>Queen</th>
<th>row</th>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{2}</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>{3}</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>{4}</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>{5}</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>{6}</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>{7}</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>{8}</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5-14 Steps leading to the first successful configuration of four queens as found by the method `putQueen()`
### Backtracking (continued)

#### Figure 5-15 Changes in the four arrays used by method putQueen()

<table>
<thead>
<tr>
<th>positionInRow</th>
<th>column</th>
<th>leftDiagonal</th>
<th>rightDiagonal</th>
<th>row</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 2, )</td>
<td>(!a, a, a, !a, a)</td>
<td>(!a, a, a, !a, a, a, a)</td>
<td>(a, a, !a, !a, a, a, a)</td>
<td>0, 1</td>
</tr>
<tr>
<td>{1}{2}</td>
<td>{1} {2}</td>
<td>{1} {2}</td>
<td>{2}{1}</td>
<td>{1}{2}</td>
</tr>
<tr>
<td>(0, 3, 1, )</td>
<td>(!a, !a, a, !a)</td>
<td>(!a, a, a, !a, !a, a, a)</td>
<td>(a, !a, a, !a, !a, a, a)</td>
<td>1, 2</td>
</tr>
<tr>
<td>{1}{3}{4}</td>
<td>{1} {4} {3}</td>
<td>{1} {4}{3}</td>
<td>{3} {1} {4}</td>
<td>{3}{4}</td>
</tr>
<tr>
<td>(1, 3, 0, 2)</td>
<td>(!a, !a, !a, !a)</td>
<td>(a, !a, a, !a, !a, a, a)</td>
<td>(a, !a, !a, a, !a, !a, a)</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>{5} {6} {7} {8}</td>
<td>{7} {5} {8} {6}</td>
<td>{5} {7} {6} {8}</td>
<td>{6} {5} {8} {7}</td>
<td>{5}{6}{7}{8}</td>
</tr>
</tbody>
</table>
Backtracking (continued)

Figure 5-16 Changes on the run-time stack for the first successful completion of putQueen()
Backtracking (continued)

Figure 5-17 Changes to the chessboard leading to the first successful configuration
Backtracking (continued)

```java
putQueen(1);
  col = 0;
  col = 1;
  col = 2;
  col = 3;
putQueen(2)
  col = 0;
  putQueen(3)
  col = 0;
  col = 1;
  col = 2;
  success;
```

Figure 5-18 Trace of calls to `putQueen()` to place four queens (continued)
Case Study: A Recursive Descent Interpreter

- The process of translating one executable statement at a time and immediately executing it is called **interpretation**
- Translating the entire program first and then executing it is called **compilation**
Case Study: A Recursive Descent Interpreter (continued)

Figure 5-19 Diagrams of methods used by the recursive descent interpreter
Case Study: A Recursive Descent Interpreter (continued)

Figure 5-19 Diagrams of methods used by the recursive descent interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

term()
   f1 = factor();
   while current token is either / or *
      f2 = factor();
      f1 = f1 * f2 or f1 / f2;
   return f1;

factor()
   process all +s and – s preceding a factor;
   if current token is an identifier
      return value assigned to the identifier;
   else if current token is a number
      return the number;
   else if current token is ( 
      e = expression();
      if current token is )
      return e;
Case Study: A Recursive Descent Interpreter (continued)

import java.io.*;

class Id {
    private String id;
    public double value;
    public Id(String s, double d) {
        id = s; value = d;
    }
    public boolean equals(Object node) {
        return id.equals(((Id)node).id);
    }
    public String toString() {
        return id + " = " + value + "; "
    }
}

**Figure 5-20 Implementation of a simple language interpreter**
Case Study: A Recursive Descent Interpreter (continued)

```java
public class Interpreter {
    private StreamTokenizer fIn = new StreamTokenizer(
        new BufferedReader(
            new InputStreamReader(System.in)));
    private java.util.LinkedList idList = new java.util.LinkedList();
    public Interpreter() {
        fIn.wordChars('$','$'); // include underscores and dollar signs as
        fIn.wordChars('_','_'); // word constituents; examples of identifiers:
            // var1, x, _pqr123xyz, $aName;
        fIn.ordinaryChar('/'); // by default, '/' is a comment character;
        fIn.ordinaryChar('.'); // otherwise "n-123.45"
        fIn.ordinaryChar('-'); // is considered a token;
    }
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

```java
private void issueError(String s) {
    System.out.println(s);
    Runtime.getRuntime().exit(-1);
}
private void addOrModify(String id, double e) {
    Id tmp = new Id(new String(id), e);
    int pos;
    if ((pos = idList.indexOf(tmp)) != -1)
        ((Id)idList.get(pos)).value = e;
    else idList.add(tmp);
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

private double findValue(String id) {
    int pos;
    if ((pos = idList.indexOf(new Id(id, 0.0))) != -1)
        return ((Id)idList.get(pos)).value;
    else issueError("Unknown variable " + id);
    return 0.0;  // this statement is never reached;
}
private double factor() throws IOException {
    double val, minus = 1.0;
    fIn.nextToken();
    while (fIn.ttype == '+' || fIn.ttype == '-') {  // take all '+'s
        if (fIn.ttype == '-')  // and '-'s;
            minus *= -1.0;
        fIn.nextToken();
    }
}

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

```java
}  
if (fIn.ttype == fIn.TT_NUMBER || fIn.ttype == '.') {
    if (fIn.ttype == fIn.TT_NUMBER) {  // factor can be a number:
        val = fIn.nval;  // 123, .123, 123., 12.3;
        fIn.nextToken();
    }
    else val = 0;
    if (fIn.ttype == '.') {
        fIn.nextToken();
        if (fIn.ttype == fIn.TT_NUMBER) {
            String s = fIn.nval + "";
            s = "." + s.substring(0,s.indexOf('.'));
            val += Double.valueOf(s).doubleValue();
        }
        else fIn.pushBack();
    }
    else fIn.pushBack();
```

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

```java
} else if (fIn.ttype == '(') { // or a parenthesized
    val = expression(); // expression,
    if (fIn.ttype == ')')
        fIn.nextToken();
    else issueError("Right parenthesis is left out.");
} else {
    val = findValue(fIn.sval); // or an identifier;
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

```java
return minus*val;
}
private double term() throws IOException {
    double f = factor();
    while (true) {
        fIn.nextToken();
        switch (fIn.ttype) {
            case '*' : f *= factor(); break;
            case '/' : f /= factor(); break;
            default : fIn.pushBack(); return f;
        }
    }
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

```java
private double expression() throws IOException {
    double t = term();
    while (true) {
        fIn.nextToken();
        switch (fIn.ttype) {
            case '+' : t += term(); break;
            case '-' : t -= term(); break;
            default : fIn.pushBack(); return t;
        }
    }
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)
public void run() {
    try {
        System.out.println("The program processes statements in the "
                + "following format:\n" + "\t<id> = <expr>;\n\tprintln <id>\n\tstatus\n\ttend");
        while (true) {
            System.out.print("Enter a statement: ");
            fIn.nextToken();
            String str = fIn.sval;
            if (str.toUpperCase().equals("STATUS")) {
                java.util.Iterator it = idList.iterator();
                while (it.hasNext())
                    System.out.println(it.next());
            } else if (str.toUpperCase().equals("PRINT")) {
                fIn.nextToken();
            }
        }
    }
}
Case Study: A Recursive Descent Interpreter (continued)

```java
str = fIn.sval;
System.out.println(str + " = " + findValue(str));
}
else if (str.toUpperCase().equals("END"))
    return;
else {
    fIn.nextToken();
    if (fIn.ttype == '=') {
        double e = expression();
        fIn.nextToken();
        fIn.nextToken();
        if (fIn.ttype != ';')
            issueError("There are some extras in the statement.");
        else addOrModify(str,e);
    }
    else issueError("'=' is missing.");
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)
Case Study: A Recursive Descent Interpreter (continued)

```java
}
} catch (IOException e) {
    e.printStackTrace();
}
}
public static void main(String args[]) {
    (new Interpreter()).run();
}
```

**Figure 5-20** Implementation of a simple language interpreter (continued)
Summary

• Recursive definitions are programming concepts that define themselves

• Recursive definitions serve two purposes:
  – Generating new elements
  – Testing whether an element belongs to a set

• Recursive definitions are frequently used to define functions and sequences of numbers
Summary (continued)

• Tail recursion is characterized by the use of only one recursive call at the very end of a method implementation.
• Backtracking is a technique for returning to a given position (e.g., entry point) after trying other avenues that are unsuccessful in solving a particular problem.
• The process of translating one executable statement at a time and immediately executing it is called interpretation.