Week 11
Recursion
(Chapter 13)

Chapter Goals

© Nicolae Popovici/iStockphoto.

- To learn to "think recursively"
- To be able to use recursive helper methods
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm
- To analyze problems that are much easier to solve by recursion than by iteration
- To process data with recursive structures using mutual recursion

Triangle Numbers

- **Recursion**: the same computation occurs repeatedly.
- Using the same method as the one in this section, you can compute the volume of a Mayan pyramid.

© Davis Mantel/iStockphoto.

- Problem: to compute the area of a triangle of width *n*
  - Assume each [] square has an area of 1
  - Also called the n^{th} *triangle number*
  - The third triangle number is 6

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

Outline of Triangle Class

```
public class Triangle
{
   private int width;

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

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;
}
```

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

Computing the area of a triangle with width 4

To find the area:

- `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
            - That method returns 1
    - The method returns `smallerArea + width = 1 + 2 = 3`
- The method returns `smallerArea + width = 3 + 3 = 6`

The method returns `smallerArea + width = 6 + 4 = 10`

Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler values.
- Two key requirements for successful recursion:
  - Every recursive call must simplify the computation in some way
  - There must be special cases to handle the simplest computations directly
- To complete our `Triangle` example, we must handle width <= 0:

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

Other Ways to Compute Triangle Numbers

- 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;
```

- Using math:

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

section_1/Triangle.java

```java
/**
    A triangular shape composed of stacked unit squares like this:
    []
    [][]
    [][][]
    ...
*/
public class Triangle
...
```

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

section_1/TriangleTester.java

```java
public class TriangleTester
{
   public static void main(String[] args)
   {
      Triangle t = new Triangle(10);
      int area = t.getArea();
      System.out.println("Area: " + area);
      System.out.println("Expected: 55");
   }
```

Program Run:

```
Area: 55
Expected: 55
```

Tracing Through Recursive Methods

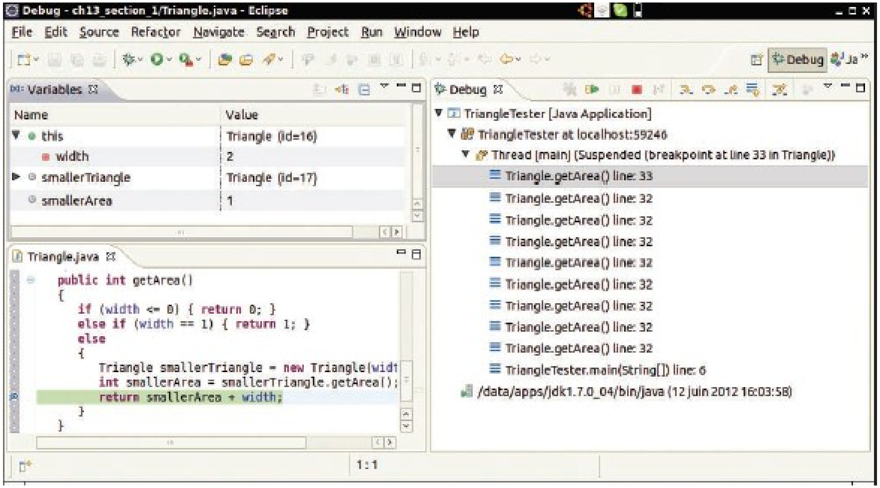

Figure 1 Debugging a Recursive Method

To debug recursive methods with a debugger, you need to be particularly careful, and watch the call stack to understand which nested call you currently are in.

Thinking Recursively

© Nikada/iStockphoto.

Thinking recursively is easy if you can recognize a subtask that is similar to the original task.

- Problem: test whether a sentence is a palindrome
- Palindrome: a string that is equal to itself when you reverse all characters
  - A man, a plan, a canal – Panama!
  - Go hang a salami, I'm a lasagna hog
  - Madam, I'm Adam

The Efficiency of Recursion: 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

section_3/RecursiveFib.java

```
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using a recursive method.
*/
public class RecursiveFib
{
   public static void main(String[] args)
   {
```

Program Run:

```
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**.

section_3/RecursiveFibTracer.java

```java
import java.util.Scanner;

/**
    This program prints trace messages that show how often the
    recursive method for computing Fibonacci numbers calls itself.
*/
public class RecursiveFibTracer
{
   public static void main(String[] args)
```

Program Run:

```
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
```

section_3/RecursiveFibTracer.java

```
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)`

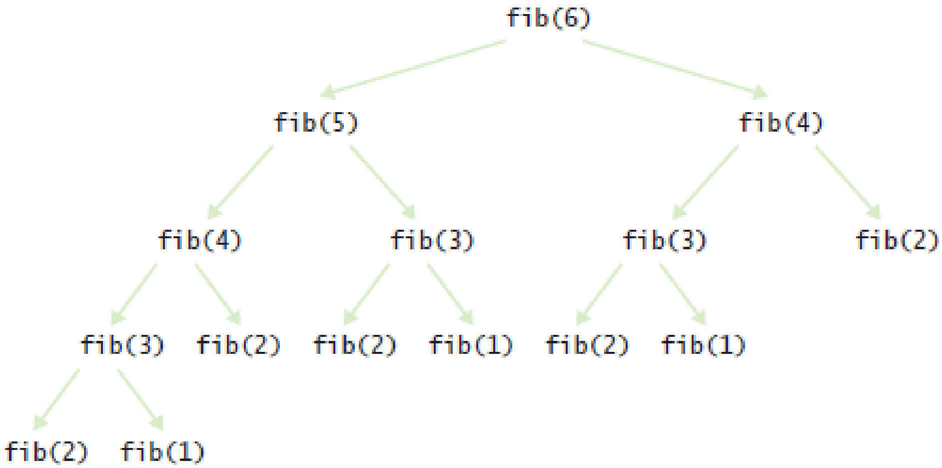

Figure 2 Call Pattern of the Recursive `fib` Method

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.

section_3/LoopFib.java

```java
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using an iterative method.
*/
public class LoopFib
{
   public static void main(String[] args)
   {
```

Program Run:

```
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

© pagadesign/iStockphoto.

- In most cases, the iterative and recursive approaches have comparable efficiency.
- 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.

Permutations

- Using recursion, you can find all arrangements of a set of objects.

© Jeanine Groenwald/iStockphoto.

- Design a class that will list all permutations of a string.

- A permutation is a rearrangement of the letters.
- The string `"eat"` has six permutations:

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

Permutations

- Problem: Generate all the permutations of "eat".
- First generate all permutations that start with the letter 'e', then 'a' then 't'.
- How do we generate the permutations that start with 'e'?
  - We need to know the permutations of the substring "at". But that’s the same problem — to generate all permutations — with a simpler input
- Prepend the letter 'e' to all the permutations you found of 'at'.
- Do the same for 'a' and 't'.
- Provide a special case for the simplest strings.
  - The simplest string is the empty string, which has a single permutation — itself.

section_4/Permutations.java

```
import java.util.ArrayList;

/**
   This program computes permutations of a string.
*/
public class Permutations
{
   public static void main(String[] args)
   {
```

Program Run:

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

Mutual Recursions

- **Problem:** to compute the value of arithmetic expressions such as:

```
3 + 4 * 5
(3 + 4) * 5
1 - (2 - (3 - (4 - 5)))
```

- Computing expression is complicated
  - `*` and `/` bind more strongly than `+` and `-`
  - Parentheses can be used to group subexpressions

Syntax Diagrams for Evaluating an Expression

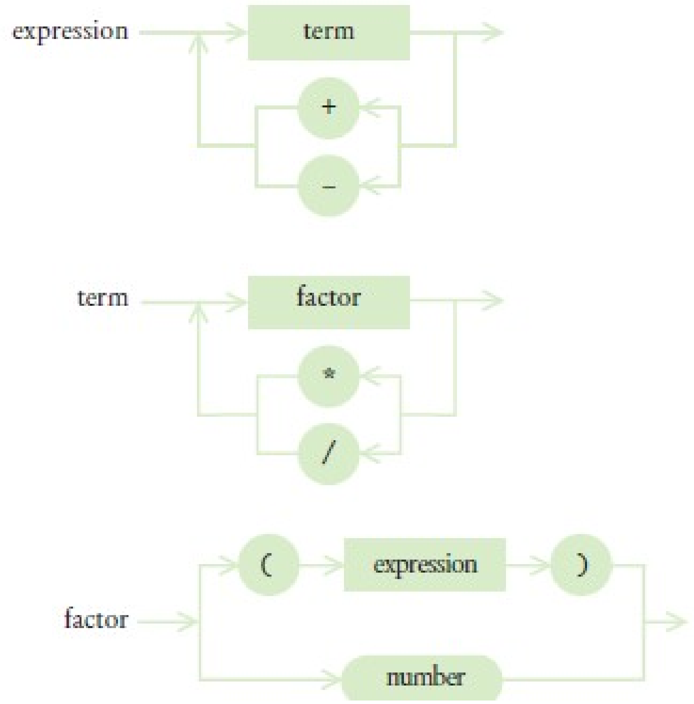

Figure 3

Mutual Recursions

- An expression can broken down into a sequence of terms, separated by + or `-` .
- Each term is broken down into a sequence of factors, separated by `*` or `/` .
- Each factor is either a parenthesized expression or a number.
- The syntax trees represent which operations should be carried out first.

Syntax Tree for Two Expressions

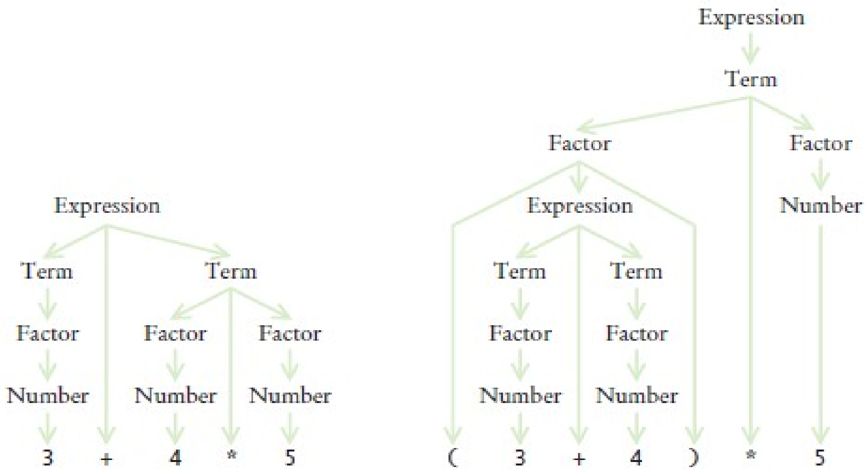

Figure 4 Syntax Trees for Two Expressions

- In a mutual recursion, a set of cooperating methods calls each other repeatedly.
- To compute the value of an expression, implement 3 methods that call each other recursively:

```
getExpressionValue
getTermValue
getFactorValue
```