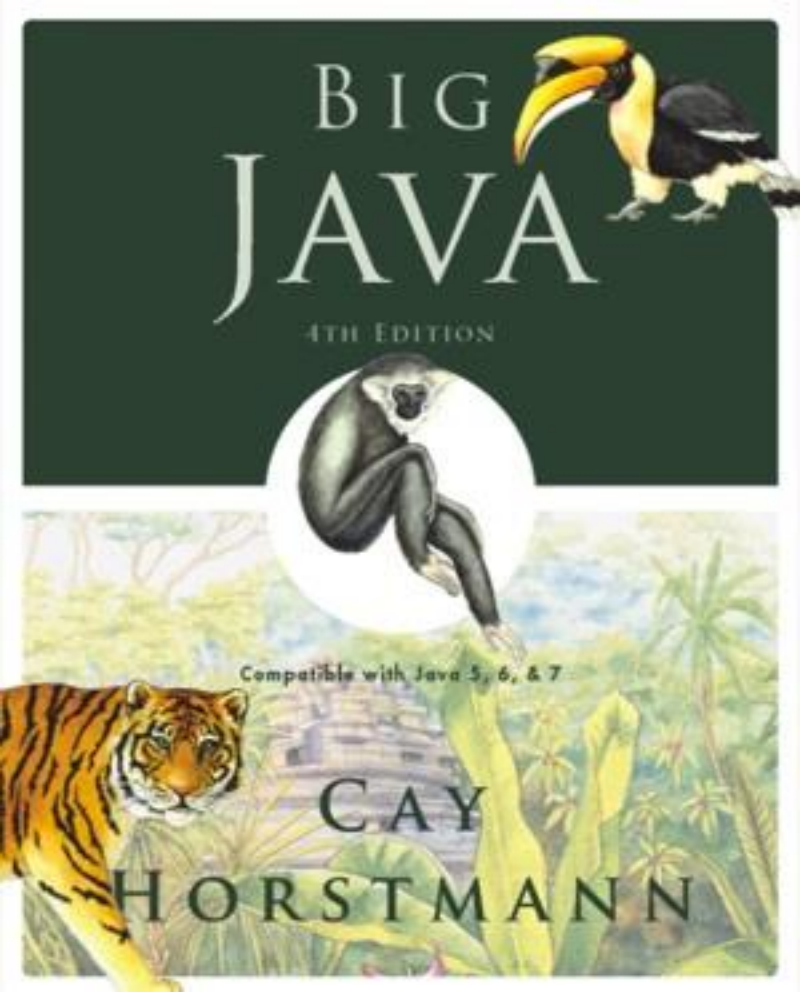


الفصل ١٣ - استدعاء ذاتي

Chapter 13 – Recursion

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

- لمعرفة المزيد عن طريقة العودية
- لفهم العلاقة بين عودية والتكرار
- لتحليل المشاكل التي هي أسهل بكثير لحل عن طريق عودية من قبل التكرار
- لمعرفة الى "التفكير بشكل متكرر"
- لتكون قادرة على استخدام أساليب المساعد متكررة
- لفهم عند استخدام العودية يؤثر على كفاءة خوارزمية

Triangle Numbers

أرقام المثلث

- 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

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

- حساب مساحة المثلث العرض N
- تحمل كل [] لديه مربع مساحة ١
- كما دعا عدد مثلث الألف
- عدد مثلث الثالث هو ٦

Outline of `Triangle` Class

الخطوط العريضة للمثلث الفئة

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

Handling Triangle of Width 1

التعامل مع مثلث العرض ١

- 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; }
   ...
}
```

- يتكون مثلث مربع واحد
- مساحتها ١
- إضافة رمز لأسلوب getArea لعرض ١

Handling the General Case

التعامل مع حالة عامة

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

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

- نفترض أننا نعرف المنطقة من أصغر، مثلث الملونة

- Area of larger triangle can be calculated as:

```
smallerArea + width
```

- ويمكن حساب مساحة المثلث أكبر على النحو التالي:

- 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();
```

- للحصول على مساحة المثلث أصغر
- جعل مثلث أصغر ويطلب منها لمنطقته:

Completed `getArea` Method

- الانتهاء أسلوب getArea

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

Computing the area of a triangle with width 4

حساب مساحة المثلث مع عرض ٤

- `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
```

- طريقة getAreaيجعل مثلث أصغر من العرض ٣
- ويدعو getAreaعلى هذا المثلث
- هذا الأسلوب يجعل مثلث أصغر من عرض ٢
- ويدعو getAreaعلى هذا المثلث
- هذا الأسلوب يجعل مثلث أصغر من العرض ١
- ويدعو getAreaعلى هذا المثلث
- أن الأسلوب بإرجاع ١
- الأسلوب بإرجاع
- =smallerArea + width ١ + ٢ = ٣
- الأسلوب بإرجاع
- =smallerArea + width ٣ + ٣ = ٦
- الأسلوب بإرجاع
- =smallerArea + width ٦ + ٤ = ١٠

Recursion

استدعاء ذاتي

- 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` example, we must handle width <= 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*

- Aحساب عودي يحل المشكلة عن طريق استخدام حل من نفس المشكلة مع القيم أبسط
- لاستدعاء ذاتي لإنهاء، يجب أن يكون هناك حالات خاصة لأبسط المدخلات
- لاستكمال مثالنا المثلث، يجب علينا التعامل مع العرض <= ٠:

- اثنين من المتطلبات الرئيسية لنجاح استدعاء ذاتي:
- كل مكالمة العودية يجب تبسيط حساب في بعض الطريق
- يجب أن تكون هناك حالات خاصة للتعامل مع أبسط العمليات الحسابية مباشرة

Other Ways to Compute Triangle Numbers

- طرق أخرى لحساب أرقام المثلث

- The area of a triangle equals the sum:

- منطقة مثلث تساوي مجموع:

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

- Using a simple loop:

- باستخدام حلقة بسيطة:

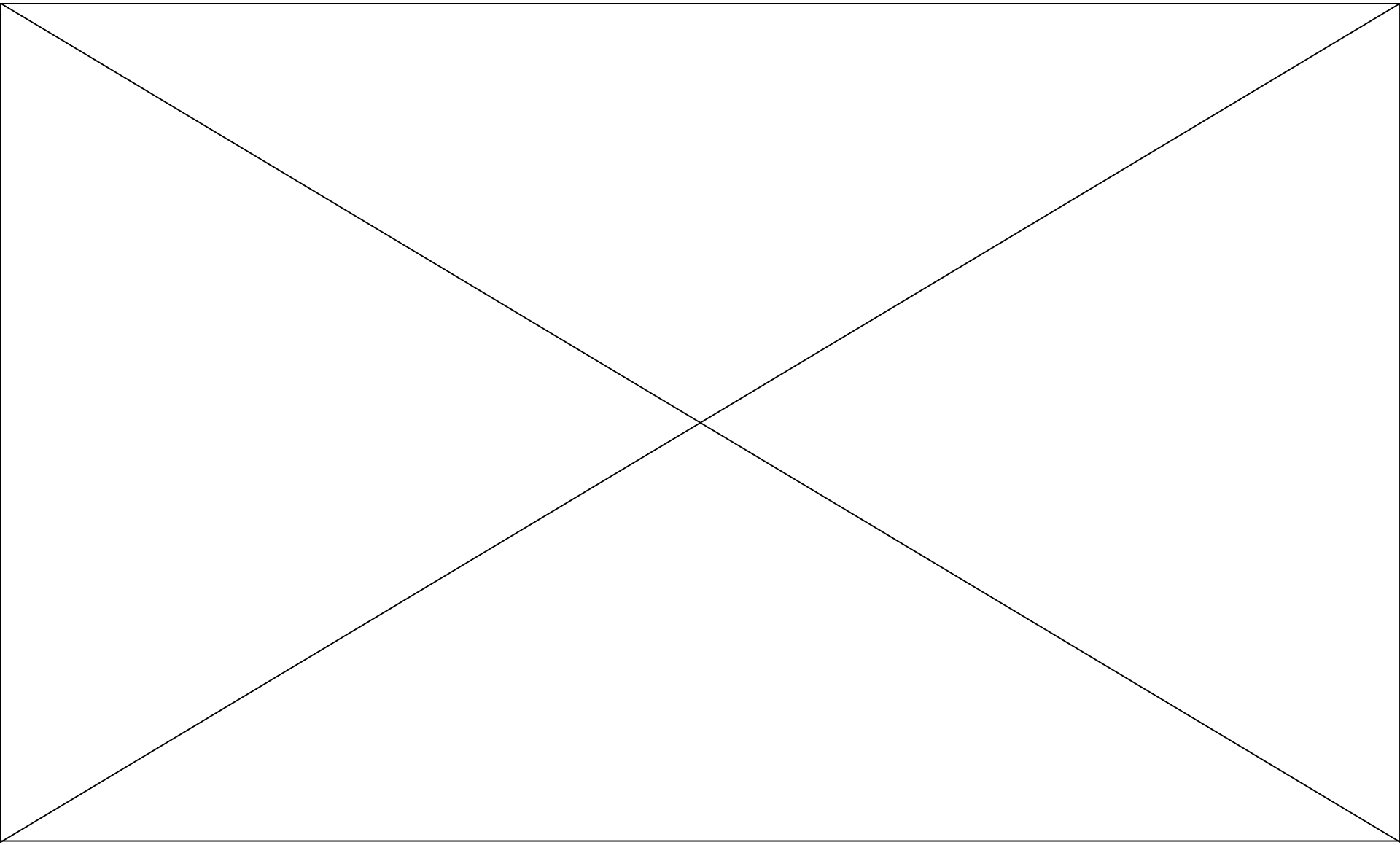
```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

- Using math:

- باستخدام الرياضيات:

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

Animation 13.1



ch13/triangle/Triangle.java

```
1  /**
2     A triangular shape composed of stacked unit squares like this:
3     []
4     [][]
5     [][][]
6     ...
7  */
8  public class Triangle
9  {
10    private int width;
11
12    /**
13       Constructs a triangular shape.
14       @param aWidth the width (and height) of the triangle
15    */
16    public Triangle(int aWidth)
17    {
18       width = aWidth;
19    }
20
```

Continued

ch13/triangle/Triangle.java (cont.)

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

ch13/triangle/TriangleTester.java

```
 1  public class TriangleTester
 2  {
 3     public static void main(String[] args)
 4     {
 5        Triangle t = new Triangle(10);
 6        int area = t.getArea();
 7        System.out.println("Area: " + area);
 8        System.out.println("Expected: 55");
 9     }
10  }
```

Program Run:

```
Enter width: 10
Area: 55
Expected: 55
```

Self Check 13.1

Why is the statement

لماذا هو البيان

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

in the `getArea` method unnecessary?

Answer: 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.

- في طريقة getAreaلا لزوم لها؟
- الإجابة: لنفترض أننا حذف بيان. عند حساب مساحة المثلث مع عرض ١، ونحن حساب مساحة المثلث مع عرض ٠ ك ٠، ثم قم بإضافة ١، للوصول إلى المكان الصحيح.

Self Check 13.2

كيف سيكون تعديل البرنامج لحساب متكرر مساحة مربع؟

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

Answer: You would compute the smaller area recursively, then return

أنت حساب مساحة أصغر بشكل متكرر، ثم العودة :

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

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

بالطبع، سيكون من الأسهل حساب

Of course, it would be simpler to compute

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

Tracing Through Recursive Methods

البحث عن المفقودين من خلال طرق تكراري

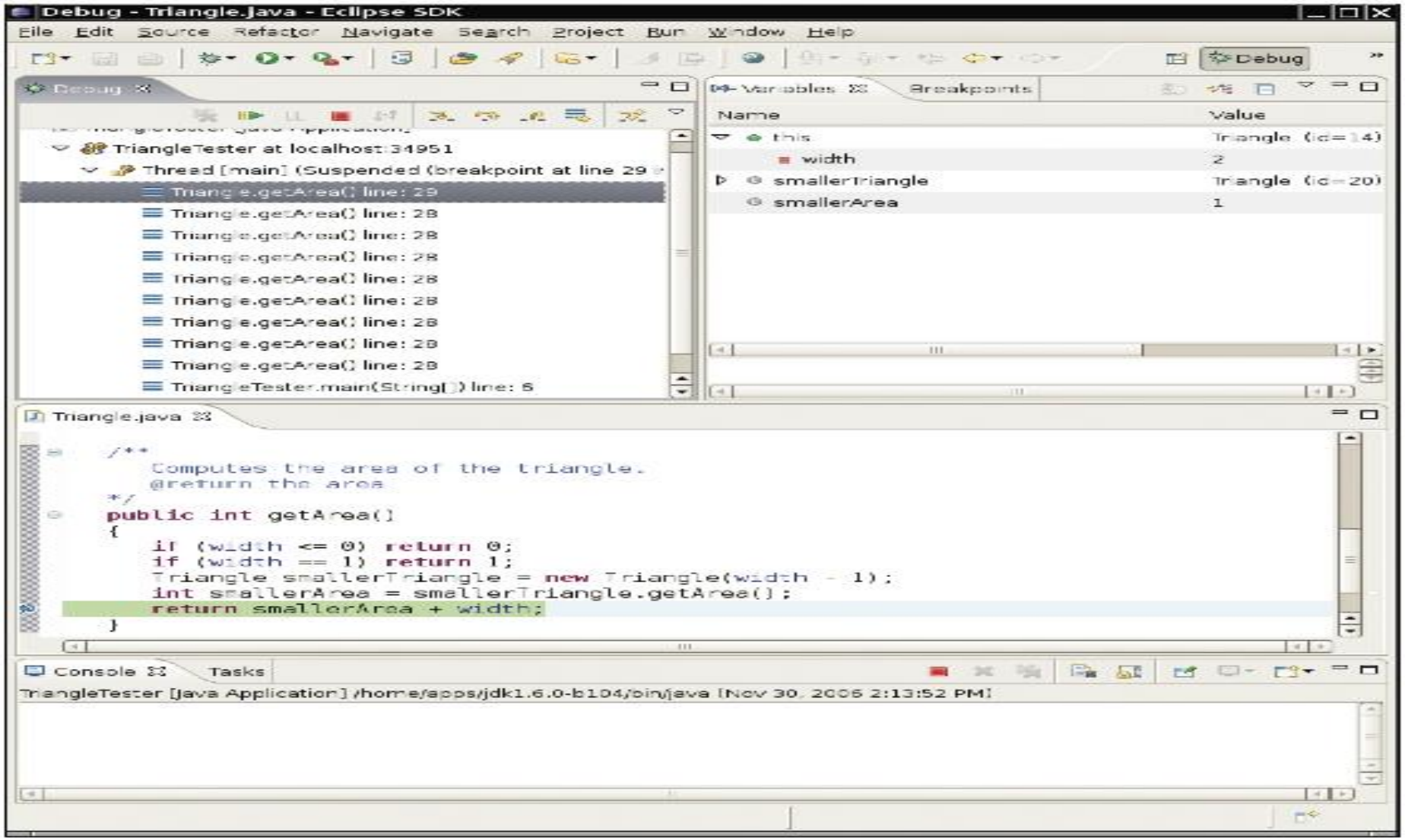


Figure 1 Debugging a Recursive Method

Thinking Recursively

التفكير بشكل متكرر

- 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*

- المشكلة: اختبار ما إذا كان الحكم هو سياق متناظر
- سياق متناظر: Aالسلسلة التي تساوي نفسها عند عكس كل الحروف
- رجل، خطة، قناة - بنما!
- العودة شنق السلامي، وأنا خنزير اللازانيا
- سيدتي، أنا آدم

Implement `isPalindrome` Method

```
public class Sentence
{
   private String text;
   /**
      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
   */
```

Continued

Implement `isPalindrome` Method (cont.)

```
public boolean isPalindrome()
   {
      ...
   }
}
```

Thinking Recursively: Step-by-Step

التفكير بشكل متكرر: الخطوة بخطوة

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*
   - *Cut the string into two halves*

١. النظر في طرق مختلفة لتبسيط المدخلات

- وهنا عدة احتمالات:
- إزالة الحرف الأول
- إزالة الحرف الأخير
- إزالة كل الأحرف الأولى والأخيرة
- إزالة حرف من الوسط
- قطع السلسلة إلى نصفين

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*
   - *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*

التفكير بشكل متكرر: الخطوة بخطوة

٢. الجمع بين الحلول مع مدخلات بساطة إلى حل المشكلة الأصلية

- معظم تبسيط واعد: إزالة الأحرف الأولى والأخيرة
- "آدم، أنا آدا" هو سياق متناظر جدا!
- وهكذا، فإن الكلمة هي سياق متناظر إذا
- تطابق الأحرف الأولى والأخيرة، و
- كلمة حصل عليها عن طريق إزالة الأحرف الأولى والأخيرة هو سياق متناظر
- ماذا لو الحرف الأول أو الأخير ليس الرسالة؟ تجاهله
- إذا كانت الأحرف الأولى والأخيرة هي حروف، تحقق ما إذا كانت تطابق. ؟ إذا كان الأمر كذلك، إزالة كل واختبار أقصر سلسلة
- إذا الحرف الأخير ليس إلكتروني، إزالته واختبار أقصر سلسلة
- إذا الحرف الأول هو لا بريد إلكتروني، إزالته واختبار أقصر سلسلة

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*

٣. البحث عن حلول لأبسط المدخلات
- سلاسل مع حرفين
- لا حالة خاصة المطلوبة؛ خطوتين لا يزال ساريا
- سلاسل بحرف واحد
- هم متناظرات
- سلسلة فارغة
- بل هو سياق متناظر

Thinking Recursively: Step-by-Step

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

٤. تنفيذ الحل من خلال الجمع بين حالات بسيطة وخطوة التخفيض

```
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));
   if (Character.isLetter(first) &&
      Character.isLetter(last))
   {
      // Both are letters.
      if (first == last)
      {
```

Continued

Thinking Recursively: Step-by-Step (cont.)

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

Continued

Thinking Recursively: Step-by-Step (cont.)

```
      // Remove first character.
      Sentence shorter = new
         Sentence(text.substring(1));
      return shorter.isPalindrome();
   }
}
```

Recursive Helper Methods

طرق مساعد متكررة

- 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

- في بعض الأحيان أنه من الأسهل لإيجاد حل للتكرارإذا قمت بإجراء تغيير طفيف على المشكلة الأصلية
- النظر في اختبار سياق متناظر من الشريحة السابقة
- فمن غير فعالة قليلا لبناء الأجسام جملة جديدة في كل خطوة

Recursive Helper Methods

- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:
- بدلا من اختبار ما إذا كانت العقوبة هي سياق متناظر، تحقق ما إذا كان فرعية هي سياق متناظر:

```
/**
   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)
```

Recursive Helper Methods

- Then, simply call the helper method with positions that test the entire string:
- ثم، لمجرد استدعاء الأسلوب المساعد مع المواقف التي اختبار السلسلة بأكملها:

```
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));
   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;
```

Continued

Recursive Helper Methods: `isPalindrome` (cont.)

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

Self Check 13.3

Do we have to give the same name to both `isPalindrome` methods?

Answer: No — the first one could be given a different name such as `substringIsPalindrome`.

- هل لدينا لإعطاء نفس الاسم على حد سواء أساليب isPalindrome؟
- الإجابة: لا - أول واحد يمكن أن يعطى اسما مختلفا مثل substringIsPalindrome.

Self Check 13.4

When does the recursive `isPalindrome` method stop calling itself?

Answer: When `start >= end`, that is, when the investigated string is either empty or has length 1.

- متى تتوقف طريقة isPalindromeعودي تطلق على نفسها؟
- الجواب: عند بدء> = نهاية، وهذا هو، عندما السلسلة التحقيق إما فارغة أو لديه طول ١.

Fibonacci Sequence

- Fibonacci sequence is a sequence of numbers defined by

متتالية فيبوناتشي هو سلسلة من الأرقام التي تحددها

$f_1 = 1$
$f_2 = 1$
$f_n = f_{n-1} + f_{n-2}$

- First ten terms:

أول عشرة حيث:

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

ch13/fib/RecursiveFib.java

```
 1  import java.util.Scanner;
 2
 3  /**
 4     This program computes Fibonacci numbers using a recursive method.
 5  */
 6  public class RecursiveFib
 7  {
 8     public static void main(String[] args)
 9     {
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        {
16           long f = fib(i);
17           System.out.println("fib(" + i + ") = " + f);
18        }
19     }
20
```

Continued

ch13/fib/RecursiveFib.java (cont.)

```
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         else return fib(n - 1) + fib(n - 2);
30      }
31   }
```

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, let's insert **trace messages**

- تنفيذ العودية أكذوبة واضح وصريح
- مشاهدة الناتج عن كثب كما تقوم بتشغيل برنامج الاختبار
- المكالمات القليلة الأولى إلى فيب سريعة جدا
- للحصول على قيم أكبر، البرنامج مؤقتا وقتا طويلا بشكل مثير للدهشة بين مخرجات
- لمعرفة المشكلة، دعونا إدراج رسائل التتبع

ch13/fib/RecursiveFibTracer.java

```
 1  import java.util.Scanner;
 2
 3  /**
 4     This program prints trace messages that show how often the
 5     recursive method for computing Fibonacci numbers calls itself.
 6  */
 7  public class RecursiveFibTracer
 8  {
 9     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     }
```

ch13/fib/RecursiveFibTracer.java (cont.)

```
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   }
```

Continued

ch13/fib/RecursiveFibTracer.java (cont.)

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
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
```

Continued

ch13/fib/RecursiveFibTracer.java (cont)

```
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)`

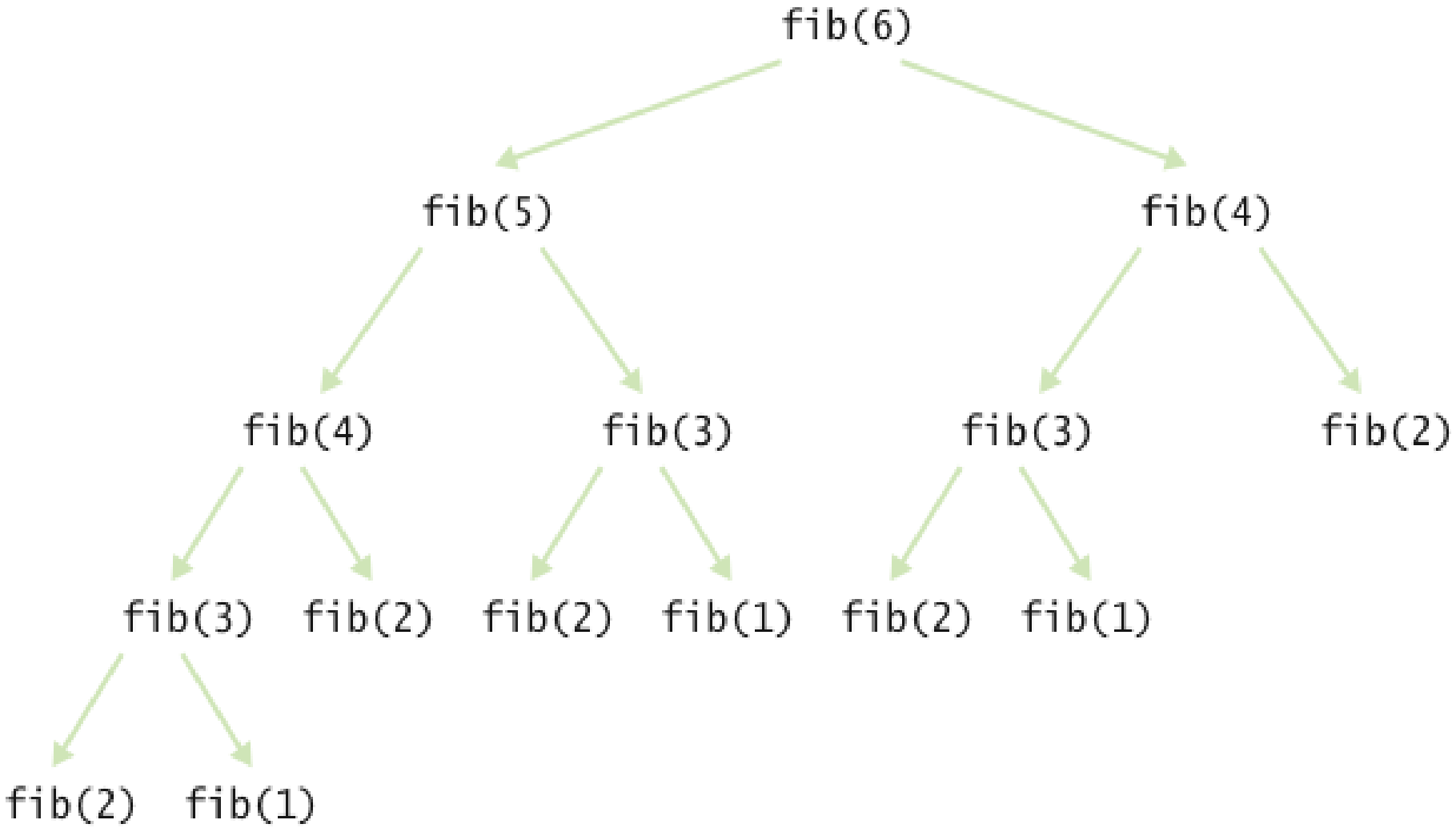


Figure 2 Call Pattern of the Recursive fib Method

The Efficiency of Recursion

كفاءة استدعاء ذاتي

- 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

- أسلوب يأخذ وقتا طويلا لأنه يحسب نفس القيم مرارا وتكرارا
- حساب فيبوناتشي (٦) يدعو أكذوبة (٣) ثلاث مرات
- تقليد عملية قلم رصاص ورقة لتجنب حساب القيم أكثر من مرة

ch13/fib/LoopFib.java

```
 1  import java.util.Scanner;
 2
 3  /**
 4     This program computes Fibonacci numbers using an iterative method.
 5  */
 6  public class LoopFib
 7  {
 8     public static void main(String[] args)
 9     {
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        {
16           long f = fib(i);
17           System.out.println("fib(" + i + ") = " + f);
18        }
19     }
20
```

Continued

ch13/fib/LoopFib.java (cont.)

```
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 olderValue = 1;
30         long oldValue = 1;
31         long newValue = 1;
32         for (int i = 3; i <= n; i++)
33         {
34            newValue = oldValue + olderValue;
35            olderValue = oldValue;
36            oldValue = newValue;
37         }
38         return newValue;
39      }
40   }
```

Continued

ch13/fib/LoopFib.java (cont.)

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

- 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

- في بعض الأحيان، وهو حل العودية يعمل بشكل أبطأ بكثير من نظيرتها تكراري
- في معظم الحالات، فإن الحل عودي فقط أبطأ قليلا
- و isPalindromeتكرارية يؤدي فقط أفضل قليلا من حل العودية
- كل استدعاء الأسلوب العودية يأخذ قدرا معينا من وقت المعالج
- يمكن المجمعين الذكية تجنب استدعاءات الأسلوب العودية إذا اتبعوا أنماط بسيطة
- معظم المجمعين لا تفعل ذلك
- في كثير من الحالات، وإيجاد حل العودية أسهل للفهم وتنفيذ بشكل صحيح من حل تكراري
- "لتكرار هو الإنسان، لعنة إلهية". .Lبيتر الألمانية

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--;
         }
```

Continued

Iterative `isPalindrome` Method (cont.)

```
         else
            return false;
      }
      if (!Character.isLetter(last))
         end--;
      if (!Character.isLetter(first))
         start++;
   }
   return true;
}
```

Self Check 13.5

Is it faster to compute the triangle numbers recursively, as shown in Section 13.1, or is it faster to use a loop that computes 1 + 2 + 3 + . . . + width?

Answer: The loop is slightly faster. Of course, it is even faster to simply compute width * (width + 1) / 2.

- هو أسرع لحساب الأرقام مثلث بشكل متكرر، كما هو مبين في القسم ١٣.١، أو هو أسرع لاستخدام حلقة التي يحسب ١ + ٢ + ٣ +. . . + العرض؟
- الجواب: الحلقة أسرع قليلا. بطبيعة الحال، فإنه هو أسرع لمجرد حساب العرض * (عرض + ١) / ٢.

Self Check 13.6

You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times ... \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?

Answer: No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.

- يمكنك حساب وظيفة عاملية إما مع حلقة، وذلك باستخدام تعريف ن! = ١ × ٢ × ... × ن، أو بشكل متكرر، وذلك باستخدام تعريف ٠! = ١ و) = !ن n! - ١)! ن ×. هو نهج العودية غير فعالة في هذه الحالة؟
- الجواب: لا، الحل عودي حوالي فعالة مثل النهج المتكرر. كلاهما يتطلب ن - ١ الضرب لحساب ن !.

Permutations

التباديل

- 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"
"tea"
"tae"
```

- تصميم الفئة التي سوف قائمة في جميع الأحوال من سلسلة
- Aالتقليب هو اعادة ترتيب الحروف
- السلسلة "أكل" لديها ستة تبديلات:

Public Interface of `PermutationGenerator`

واجهة العامة

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

ch13/permute/PermutationGeneratorDemo.java

```
1  import java.util.ArrayList;
2
3  /**
4     This program demonstrates the permutation generator.
5  */
6  public class PermutationGeneratorDemo
7  {
8     public static void main(String[] args)
9     {
10    PermutationGenerator generator = new PermutationGenerator("eat");
11       ArrayList<String> permutations = generator.getPermutations();
12       for (String s : permutations)
13       {
14          System.out.println(s);
15       }
16    }
17 }
18
```

Continued

ch13/permute/PermutationGeneratorDemo.java (cont.)

Program Run:

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

To Generate All Permutations

- Generate all permutations that start with `'e'`, then `'a'`, then `'t'`
- To generate permutations starting with `'e'`, we need to find all permutations of `"at"`
- This is the same problem with simpler inputs
- Use recursion

- تولد كل التباديل التي تبدأ مع `'e'` ثم `'a'` ثم `'t'`
- لتوليد التباديل التي تبدأ بحرف `'e'` ونحن بحاجة للعثور على جميع التباديل `"at"`
- هذا هو نفس المشكلة مع مدخلات أبسط
- استخدام استدعاء ذاتي

To Generate All Permutations

لتوليد جميع التباديل

- `getPermutations`: Loop through all positions in the word to be permuted
  - حلقة من خلال جميع المناصب في كلمة للمبدل
- For each position, compute the shorter word obtained by removing i^{th} letter:
  - لكل منصب، حساب كلمة أقصر الحصول عليها عن طريق إزالة إلكتروني إيث:

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

- Construct a permutation generator to get permutations of the shorter word:
  - بناء مولد التقليب للحصول على التباديل كلمة قصيرة:

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

To Generate All Permutations

- Finally, add the removed letter to front of all permutations of the shorter word:
- أخيرا، إضافة حرف إلى إزالتها أمام كل التباديل كلمة قصيرة:

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

- Special case: Simplest possible string is the empty string; single permutation, itself
- حالة خاصة: أبسط الممكنة السلسلة سلسلة فارغة. التقليب واحد، في حد ذاته

ch13/permute/PermutationGenerator.java

```
1  import java.util.ArrayList;
2
3  /**
4     This class generates permutations of a word.
5  */
6  public class PermutationGenerator
7  {
8     private String word;
9
10    /**
11       Constructs a permutation generator.
12       @param aWord the word to permute
13    */
14    public PermutationGenerator(String aWord)
15    {
16       word = aWord;
17    }
18
```

Continued

ch13/permute/PermutationGenerator.java (cont.)

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

Continued

ch13/permute/PermutationGenerator.java (cont.)

```
33          // Loop through all character positions
34          for (int i = 0; i < word.length(); i++)
35          {
36             // Form a simpler word by removing the ith character
37             String shorterWord = word.substring(0, i)
38                   + word.substring(i + 1);
39
40             // Generate all permutations of the simpler word
41             PermutationGenerator shorterPermutationGenerator
42                   = new PermutationGenerator(shorterWord);
43             ArrayList<String> shorterWordPermutations
44                   = shorterPermutationGenerator.getPermutations();
45
46             // Add the removed character to the front of
47             // each permutation of the simpler word,
48             for (String s : shorterWordPermutations)
49             {
50                permutations.add(word.charAt(i) + s);
51             }
52          }
53          // Return all permutations
54          return permutations;
55       }
56    }
```

Self Check 13.7

What are all permutations of the four-letter word `beat`?

Answer: 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`.

- ما هي كل التباديل من أربعة أحرف كلمة فاز `beat`؟
- الجواب: يتم `b` يليهم ستة تبديلات من`eat`،`e`
  تليها ستة تبديلات من `bat`،
  وتليها ستة تبديلات من `bet`،`a`
  و`r` تليها ستة تبديلات من `bea`.

Self Check 13.8

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?

Answer: Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

- لدينا استدعاء ذاتي لمولد التقليب يتوقف عند سلسلة فارغة. ما تعديل بسيط من شأنه أن يجعل وقف استدعاء ذاتي في سلاسل من طول ٠ أو ١؟
- الجواب: ببساطة تغيير إذا (word.length () == 0)للو ((word.length () <= 1، وذلك لأن كلمة بحرف واحد هو أيضا التقليب لتقديرها.

Self Check 13.9

Why isn't it easy to develop an iterative solution for the permutation generator?

Answer: 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 — see Exercise P13.12.

- لماذا ليس من السهل وضع حل تكرارية لمولد التقليب؟
- الإجابة: ومن شأن حل تكرارية يكون حلقة التي يحسب التقليب القادم من سابقاتها الجسم. ولكن لا توجد آلية واضحة للحصول على التقليب المقبل. على سبيل المثال، إذا كنت بالفعل تم العثور عليها التباديل `eat`, `eta`, and `aet`, فإنه ليس من الواضح كيفية استخدام هذه المعلومات للحصول على التقليب المقبل. في الواقع، هناك آلية مبتكرة للقيام بذلك فقط، ولكنها أبعد ما تكون عن الوضوح - راجع التمرين P13.12.

The Limits of Computation

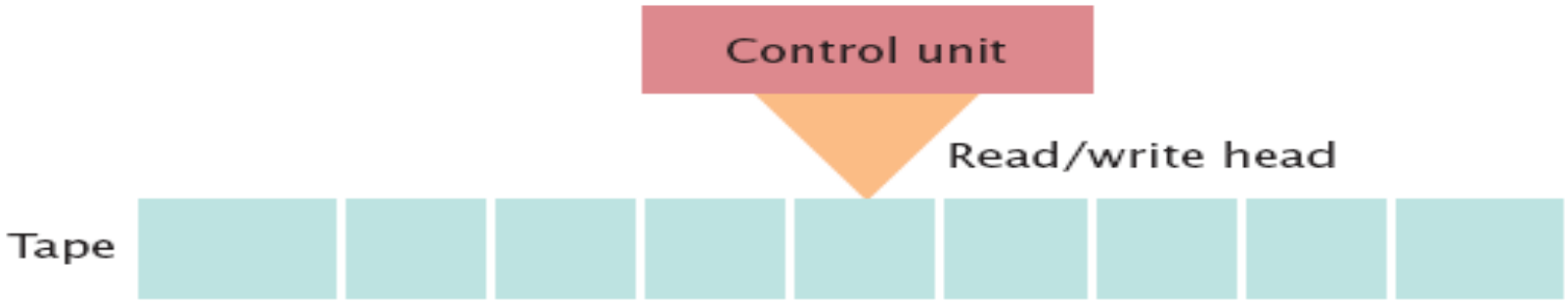
حدود الحساب

Alan Turing

The Limits of Computation

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
1	1	1	left	4
2	0	0	right	2
2	1	1	right	2
2	2	0	left	3
3	0	0	left	3
3	1	1	left	3
3	2	2	right	1
4	1	1	right	5
4	2	0	left	4



A Turing Machine

Using Mutual Recursions

باستخدام استدعاء ذاتي المتبادلة

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

- المشكلة: لحساب قيمة تعبيرات حسابية مثل

```
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*

- الحوسبة التعبير غير معقدة
- * و/ ربط بقوة أكبر من + و-
- قوسين يمكن استخدامها لمجموعة التعابير الجزئيه

Syntax Diagrams for Evaluating an Expression

تركيب رسم تخطيطي لتقييم تعبير

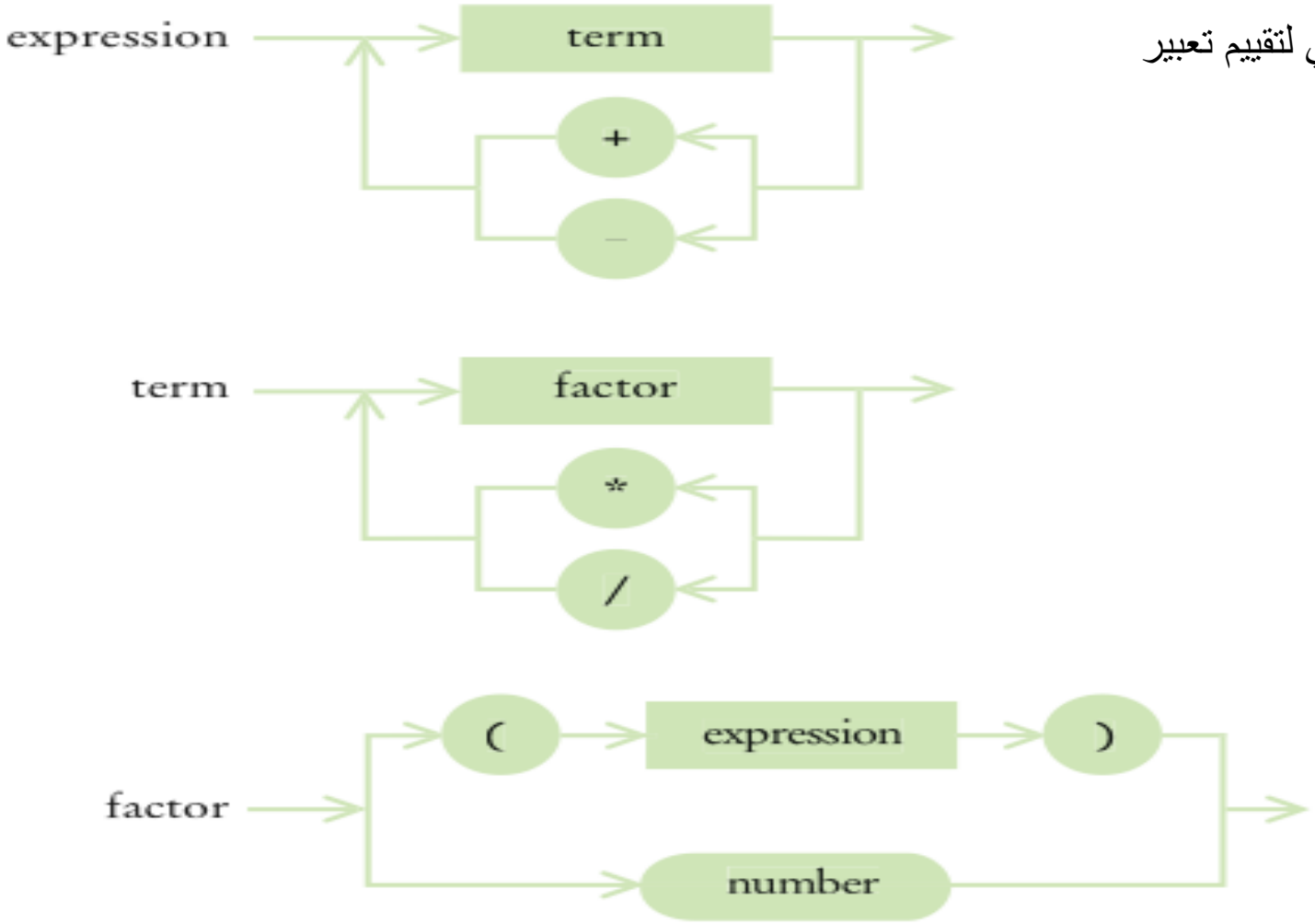


Figure 3 Syntax Diagrams for Evaluating an Expression

Using 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

- تعبير يمكن تقسيمها إلى سلسلة من الشروط، مفصولة + أو -
- يتم تقسيم كل فصل دراسي أسفل إلى سلسلة من العوامل، مفصولة * أو /
- كل عامل هو إما التعبير بالأقواس أو عدد
- تمثل أشجار في بناء الجملة التي يجب أن تتم عمليات أولا

Syntax Tree for Two Expressions

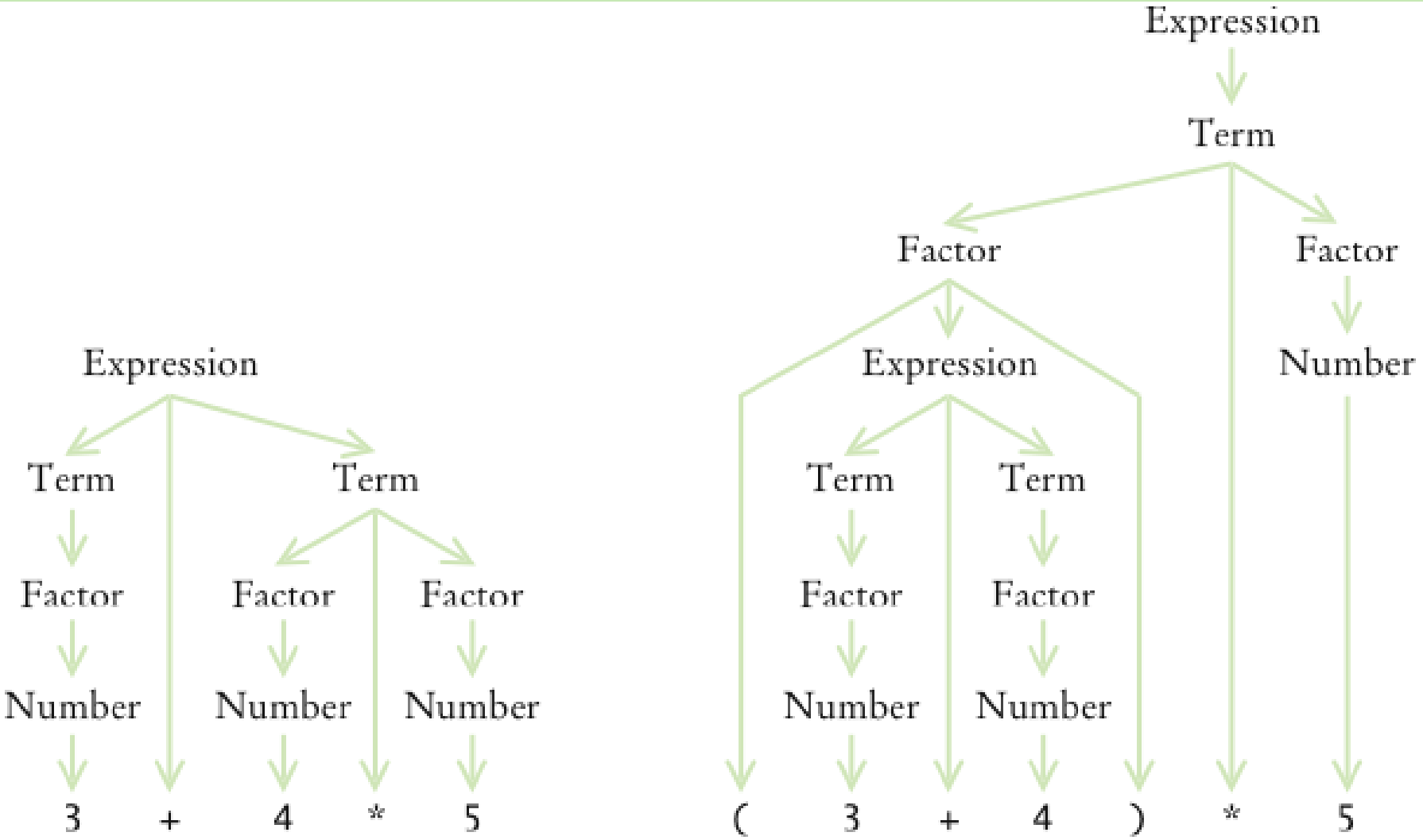


Figure 4 Syntax Trees for Two Expressions

Mutually Recursive Methods

طرق تكراري متبادل

- 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`

- في استدعاء ذاتي المتبادلة، ومجموعة من أساليب التعاون يدعو بعضها البعض بشكل متكرر
- لحساب قيمة تعبير، تنفيذ ٣ طرق التي تدعو بعضها البعض بشكل متكرر:
  - `getExpressionValue`
  - `getTermValue`
  - `getFactorValue`

The `getExpressionValue` Method

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Discard "+" or "-"
         int value2 = getTermValue();
         if ("+".equals(next)) value = value + value2;
         else value = value - value2;
      }
      else done = true;
   }
   return value;
}
```

The `getTermValue` Method

- The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values

- استدعاء الأسلوب getTermValue getFactorValueفي نفس الطريق، وضرب أو قسمة قيم معامل

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

The `getFactorValue` Method

```
public int getFactorValue()
{
   int value;
   String next =
   tokenpublic int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Discard "("
      value = getExpressionValue();
      tokenizer.nextToken(); // Discard ")"
   }
   else
      value = Integer.parseInt(tokenizer.nextToken());
   return value;
}
```

Using Mutual Recursions

To see the mutual recursion clearly, trace through the expression `(3+4)*5`:

لرؤية استدعاء ذاتي المتبادلة بشكل واضح، وتتبع من خلال التعبير

- `getExpressionValue` calls `getTermValue`
  - `getTermValue` calls `getFactorValue`
    - `getFactorValue` consumes the `(` input
    - `getFactorValue` calls `getExpressionValue`
      - `getExpressionValue` returns eventually with the value of `7`, having consumed `3 + 4`. This is the recursive call.
    - `getFactorValue` consumes the `)` input
    - `getFactorValue` returns `7`
  - `getTermValue` consumes the inputs `*` and `5` and returns `35`
- `getExpressionValue` returns `35`

ch13/expr/Evaluator.java

```
1  /**
2     A class that can compute the value of an arithmetic expression.
3  */
4  public class Evaluator
5  {
6     private ExpressionTokenizer tokenizer;
7
8     /**
9        Constructs an evaluator.
10       @param anExpression a string containing the expression
11       to be evaluated
12    */
13    public Evaluator(String anExpression)
14    {
15       tokenizer = new ExpressionTokenizer(anExpression);
16    }
17
```

Continued

ch13/expr/Evaluator.java (cont.)

```
18      /**
19         Evaluates the expression.
20         @return the value of the expression.
21      */
22      public int getExpressionValue()
23      {
24         int value = getTermValue();
25         boolean done = false;
26         while (!done)
27         {
28            String next = tokenizer.peekToken();
29            if ("+".equals(next) || "-".equals(next))
30            {
31               tokenizer.nextToken(); // Discard "+" or "-"
32               int value2 = getTermValue();
33               if ("+".equals(next)) { value = value + value2; }
34               else { value = value - value2; }
35            }
36            else
37            {
38               done = true;
39            }
40         }
41         return value;
42      }
43
```

Continued

ch13/expr/Evaluator.java (cont.)

```
44      /**
45         Evaluates the next term found in the expression.
46         @return the value of the term
47      */
48      public int getTermValue()
49      {
50         int value = getFactorValue();
51         boolean done = false;
52         while (!done)
53         {
54            String next = tokenizer.peekToken();
55            if ("*".equals(next) || "/".equals(next))
56            {
57               tokenizer.nextToken();
58               int value2 = getFactorValue();
59               if ("*".equals(next)) { value = value * value2; }
60               else { value = value / value2; }
61            }
62            else
63            {
64               done = true;
65            }
66         }
67         return value;
68      }
69
```

Continued

ch13/expr/Evaluator.java (cont.)

```
70      /**
71         Evaluates the next factor found in the expression.
72         @return the value of the factor
73      */
74      public int getFactorValue()
75      {
76         int value;
77         String next = tokenizer.peekToken();
78         if ("(".equals(next))
79         {
80            tokenizer.nextToken(); // Discard "("
81            value = getExpressionValue();
82            tokenizer.nextToken(); // Discard ")"
83         }
84         else
85         {
86            value = Integer.parseInt(tokenizer.nextToken());
87         }
88         return value;
89      }
90   }
```

ch13/expr/ExpressionTokenizer.java

```
1  /**
2     This class breaks up a string describing an expression
3     into tokens: numbers, parentheses, and operators.
4  */
5  public class ExpressionTokenizer
6  {
7     private String input;
8     private int start; // The start of the current token
9     private int end; // The position after the end of the current token
10
11    /**
12       Constructs a tokenizer.
13       @param anInput the string to tokenize
14    */
15    public ExpressionTokenizer(String anInput)
16    {
17       input = anInput;
18       start = 0;
19       end = 0;
20       nextToken(); // Find the first token
21    }
22
```

Continued

ch13/expr/ExpressionTokenizer.java (cont.)

```
23      /**
24         Peeks at the next token without consuming it.
25         @return the next token or null if there are no more tokens
26      */
27      public String peekToken()
28      {
29         if (start >= input.length()) { return null; }
30         else { return input.substring(start, end); }
31      }
32
```

Continued

ch13/expr/ExpressionTokenizer.java (cont.)

```
33      /**
34         Gets the next token and moves the tokenizer to the following token.
35         @return the next token or null if there are no more tokens
36      */
37      public String nextToken()
38      {
39         String r = peekToken();
40         start = end;
41         if (start >= input.length()) { return r; }
42         if (Character.isDigit(input.charAt(start)))
43         {
44            end = start + 1;
45            while (end < input.length()
46                  && Character.isDigit(input.charAt(end)))
47            {
48               end++;
49            }
50         }
51         else
52         {
53            end = start + 1;
54         }
55         return r;
56      }
57   }
```

ch13/expr/ExpressionCalculator.java

```
 1  import java.util.Scanner;
 2
 3  /**
 4     This program calculates the value of an expression
 5     consisting of numbers, arithmetic operators, and parentheses.
 6  */
 7  public class ExpressionCalculator
 8  {
 9     public static void main(String[] args)
10     {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter an expression: ");
13        String input = in.nextLine();
14        Evaluator e = new Evaluator(input);
15        int value = e.getExpressionValue();
16        System.out.println(input + "=" + value);
17     }
18  }
```

Program Run:

```
Enter an expression: 3+4*5
3+4*5=23
```

Self Check 13.10

What is the difference between a term and a factor? Why do we need both concepts?

Answer: Factors are combined by multiplicative operators (`*` and `/`), terms are combined by additive operators (`+`, `-`). We need both so that multiplication can bind more strongly than addition.

- ما هو الفرق بين مصطلح وعامل؟ لماذا نحتاج إلى كلا المفهومين؟
- الجواب: يتم الجمع بين العوامل من قبل المشغلين المضاعف (* و /)، يتم الجمع بين الشروط من قبل المشغلين المضافة (+، -). نحن بحاجة إلى كل ذلك أن الضرب يمكن ربط بقوة أكبر من الإضافة.

Self Check 13.12

Why does the expression parser use mutual recursion?

Answer: To handle parenthesized expressions, such as `2 + 3 * (4 + 5)`. The subexpression `4 + 5` is handled by a recursive call to `getExpressionValue`.

- لماذا محلل التعبير استخدام العودية المتبادلة؟
- الإجابة: للتعامل مع تعبيرات بالأقواس، مثل ٢ + ٣ * (٤ + ٥). تتم معالجة التعبير الجزئي ٤ + ٥ عن طريق مكالمة عودي إلى getExpressionValue.

Self Check 13.11

What happens if you try to parse the illegal expression `3 + 4 * ) 5`? Specifically, which method throws an exception?

Answer: The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string `")"`.

- ماذا يحدث إذا حاولت تحليل التعبير غير قانوني ٣ + ٤ *) ٥؟ على وجه التحديد، الأسلوب الذي يطرح استثناء؟
- الإجابة: إن الدعوة Integer.parseInt في getFactorValue رميات استثناء عند حصولها على السلسلة ")".