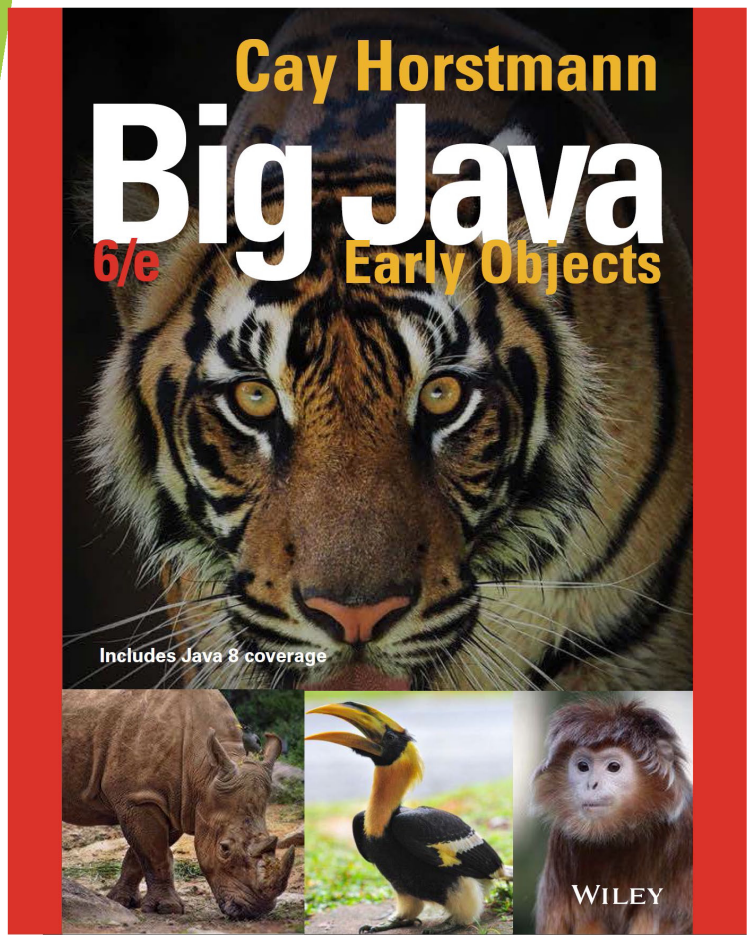


Chapter 13 : Recursion

Part I

Triangle Numbers

- Compute the area of an isosceles right triangle of
  - edge-size *n*
  - and made of `[]` squares.
- Assume each `[]` square has an area of 1
- The area is called the *n*th *triangle number*
- The third triangle number is 6 because:

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

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;
}
```

Computing the area of a triangle with width 4

- `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
- 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*

See sample code
ch13_recursion/triangle

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

Self Check 13.1

Why is the statement

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

in the `getArea` method necessary?

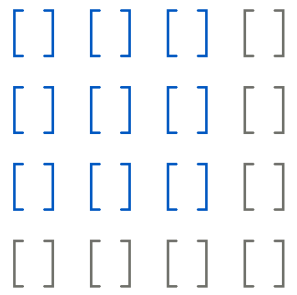
Answer: Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1, to arrive at the correct area.

Self Check 13.2

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

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

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



Of course, it would be simpler to compute

Thinking Recursively

- 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*

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

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*

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*

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*

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));
   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

- 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

Recursive Helper Methods

- 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)
```

Recursive Helper Methods

- 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));
   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`.

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.