**Section 18.1 Introduction**

• A recursive method (p. [777](http://proquest.safaribooksonline.com/9780133813036/ch18_html#page_777)) calls itself directly or indirectly through another method.

#### Section 18.2 Recursion Concepts

• When a recursive method is called (p. [778](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec2_html#page_778)) to solve a problem, it can solve only the simplest case(s), or base case(s). If called with a base case (p. [778](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec2_html#page_778)), the method returns a result.

• If a recursive method is called with a more complex problem than a base case, it typically divides the problem into two conceptual pieces—a piece that the method knows how to do and a piece that it does not know how to do.

• To make recursion feasible, the piece that the method does not know how to do must resemble the original problem, but be a slightly simpler or smaller version of it. Because this new problem resembles the original problem, the method calls a fresh copy of itself to work on the smaller problem—this is called the recursion step (p. [778](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec2_html#page_778)).

• For recursion to eventually terminate, each time a method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case. When, the method recognizes the base case, it returns a result to the previous copy of the method.

• A recursive call may be a call to another method, which in turn makes a call back to the original method. Such a process still results in a recursive call to the original method. This is known as an indirect recursive call or indirect recursion (p. [778](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec2_html#page_778)).

#### Section 18.3 Example Using Recursion: Factorials

• Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case can cause infinite recursion (p. [781](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec3_html#page_781)), eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

#### Section 18.5 Example Using Recursion: Fibonacci Series

• The Fibonacci series (p. [783](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec5_html#page_783)) begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the preceding two.

• The ratio of successive Fibonacci numbers converges on a constant value of 1.618..., a number that has been called the golden ratio or the golden mean (p. [783](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec5_html#page_783)).

• Some recursive solutions, such as Fibonacci, result in an “explosion” of method calls.

#### Section 18.6 Recursion and the Method-Call Stack

• The executing method is always the one whose stack frame is at the top of the stack, and the stack frame for that method contains the values of its local variables.

#### Section 18.7 Recursion vs. Iteration

• Both iteration and recursion are based on a control statement: Iteration uses a repetition statement, recursion a selection statement.

• Both iteration and recursion involve repetition: Iteration explicitly uses a repetition statement, whereas recursion achieves repetition through repeated method calls.

• Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails, recursion when a base case is recognized.

• Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail, whereas recursion keeps producing simpler versions of the original problem until the base case is reached.

• Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false, whereas infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

• Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls.

• Any problem that can be solved recursively can also be solved iteratively.

• A recursive approach is normally preferred over an iterative approach when it 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 few lines of code, but a corresponding iterative approach might take a large amount of code. Another reason to choose a recursive solution is that an iterative solution might not be apparent.

#### Section 18.9 Fractals

• A fractal (p. [791](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec8_html#page_791)) is a geometric figure that is generated from a pattern repeated recursively an infinite number of times.

• Fractals have a self-similar property (p. [791](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec8_html#page_791))—subparts are reduced-size copies of the whole.

#### Section 18.10 Recursive Backtracking

• In recursive backtracking (p. [802](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec10_html#page_802)), if one set of recursive calls does not result in a solution to the problem, the program backs up to the previous decision point and makes a different decision, often resulting in another set of recursive calls.

### Self-Review Exercises

[**18.1**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec14_html#ch18ans1) State whether each of the following is true or false. If false, explain why.

a) A method that calls itself indirectly is not an example of recursion.

b) Recursion can be efficient in computation because of reduced memory-space usage.

c) When a recursive method is called to solve a problem, it actually is capable of solving only the simplest case(s), or base case(s).

d) To make recursion feasible, the recursion step in a recursive solution must resemble the original problem, but be a slightly larger version of it.

[**18.2**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec14_html#ch18ans2) A \_\_\_\_\_\_\_\_\_\_ is needed to terminate recursion.

a) recursion step

b) break statement

c) void return type

d) base case

[**18.3**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec14_html#ch18ans3) The first call to invoke a recursive method is \_\_\_\_\_\_\_\_\_\_.

a) not recursive

b) recursive

c) the recursion step

d) none of the above

[**18.4**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec14_html#ch18ans4) Each time a fractal’s pattern is applied, the fractal is said to be at a new \_\_\_\_\_\_\_\_\_\_.

a) width

b) height

c) level

d) volume

[**18.5**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec14_html#ch18ans5) Iteration and recursion each involve a \_\_\_\_\_\_\_\_\_\_.

a) repetition statement

b) termination test

c) counter variable

d) none of the above

[**18.6**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec14_html#ch18ans6) Fill in the blanks in each of the following statements:

a) The ratio of successive Fibonacci numbers converges on a constant value of 1.618..., a number that has been called the \_\_\_\_\_\_\_\_\_\_ or the \_\_\_\_\_\_\_\_\_\_.

b) Iteration normally uses a repetition statement, whereas recursion normally uses a(n) \_\_\_\_\_\_\_\_\_\_ statement.

c) Fractals have a(n) \_\_\_\_\_\_\_\_\_\_ property—when subdivided into parts, each is a reduced-size copy of the whole.

### Answers to Self-Review Exercises

[**18.1**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec13_html#ch18que1)

a) False. A method that calls itself in this manner is an example of indirect recursion.

b) False. Recursion can be inefficient in computation because of multiple method calls and memory-space usage.

c) True.

d) False. To make recursion feasible, the recursion step in a recursive solution must resemble the original problem, but be a slightlysmaller version of it.

[**18.2**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec13_html#ch18que2) d

[**18.3**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec13_html#ch18que3) a

[**18.4**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec13_html#ch18que4) c

[**18.5**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec13_html#ch18que5) b

[**18.6**](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec13_html#ch18que6)

a) golden ratio, golden mean.

b) selection.

c) self-similar.

### Exercises

**18.7** What does the following code do?

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p0805pro01a)

**1**   public int mystery(int a, int b)  
 **2**   {  
 **3**      if (b == 1)  
 **4**         return a;  
 **5**      else  
 **6**         return a + mystery(a, b - 1);  
 **7**   }

**18.8** Find the error(s) in the following recursive method, and explain how to correct it (them). This method should find the sum of the values from 0 to n.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p0805pro02a)

**1**   public int sum(int n)  
 **2**   {  
 **3**      if (n == 0)  
 **4**         return 0;  
 **5**      else  
 **6**         return n + sum(n);  
 **7**   }

**18.9 (Recursive** ***power*** **Method)** Write a recursive method power(base, exponent) that, when called, returns

baseexponent

For example, power(3,4) = 3 \* 3 \* 3 \* 3. Assume that exponent is an integer greater than or equal to 1. Hint: The recursion step should use the relationship

baseexponent*=*base*·*baseexponent - 1

and the terminating condition occurs when exponent is equal to 1, because

base1*=*base

Incorporate this method into a program that enables the user to enter the base and exponent.

**18.10 (Visualizing Recursion)** It’s interesting to watch recursion “in action.” Modify the factorial method in [Fig. 18.3](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec3_html#ch18fig03) to print its local variable and recursive-call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that makes it easier to understand recursion. You may want to add such display capabilities to other recursion examples and exercises throughout the text.

**18.11 (Greatest Common Divisor)** The greatest common divisor of integers x and y is the largest integer that evenly divides into both x and y. Write a recursive method gcd that returns the greatest common divisor of x and y. The gcd of x and y is defined recursively as follows: If y is equal to 0, then gcd(x, y) is x; otherwise, gcd(x, y) is gcd(y, x % y), where % is the remainder operator. Use this method to replace the one you wrote in the application of [Exercise 6.27](http://proquest.safaribooksonline.com/9780133813036/ch06lev1sec18_html#ch06que27).

**18.12** What does the following program do?

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p0806pro01a)

**1**   // Exercise  18.12 Solution: MysteryClass.java  
 **2**   public class MysteryClass  
 **3**   {  
 **4**      public static int mystery(int[] array2, int size)  
 **5**      {  
 **6**         if (size == 1)  
 **7**            return array2[0];  
 **8**         else  
 **9**            return array2[size - 1] + mystery(array2, size - 1);  
**10**      }  
**11**   
**12**      public static void main(String[] args)  
**13**      {  
**14**         int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
**15**   
**16**         int result = mystery(array, array.length);  
**17**         System.out.printf("Result is: %d%n", result);  
**18**      } // end method main  
**19**   } // end class MysteryClass

**18.13** What does the following program do?

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p0806pro02a)

**1**   // Exercise  18.13 Solution: SomeClass.java  
 **2**   public class SomeClass  
 **3**   {  
 **4**      public static String someMethod(int[] array2, int x)  
 **5**      {  
 **6**         if (x < array2.length)  
 **7**            return String.format(  
 **8**               "%s%d ", someMethod(array2, x + 1), array2[x]);  
 **9**         else  
**10**            return "";  
**11**      }  
**12**   
**13**      public static void main(String[] args)  
**14**      {  
**15**         int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
**16**         String results = someMethod(array, 0);  
**17**         System.out.println(results);  
**18**      }  
**19**   } // end class SomeClass

**18.14 (Palindromes)** A palindrome is a string that is spelled the same way forward and backward. Some examples of palindromes are “radar,” “able was i ere i saw elba” and (if spaces are ignored) “a man a plan a canal panama.” Write a recursive methodtestPalindrome that returns boolean value true if the string stored in the array is a palindrome and false otherwise. The method should ignore spaces and punctuation in the string.

**18.15 (Eight Queens)** A puzzler for chess buffs is the Eight Queens problem, which asks: Is it possible to place eight queens on an empty chessboard so that no queen is “attacking” any other (i.e., no two queens are in the same row, in the same column or along the same diagonal)? For instance, if a queen is placed in the upper-left corner of the board, no other queens could be placed in any of the marked squares shown in [Fig. 18.20](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18fig20). Solve the problem recursively. [Hint: Your solution should begin with the first column and look for a location in that column where a queen can be placed—initially, place the queen in the first row. The solution should then recursively search the remaining columns. In the first few columns, there will be several locations where a queen may be placed. Take the first available location. If a column is reached with no possible location for a queen, the program should return to the previous column, and move the queen in that column to a new row. This continuous backing up and trying new alternatives is an example of recursive backtracking.]

**Fig. 18.20** | Squares eliminated by placing a queen in the upper-left corner of a chessboard.

**18.16 (Print an Array)** Write a recursive method printArray that displays all the elements in an array of integers, separated by spaces.

**18.17****(Print an Array Backward)** Write a recursive method stringReverse that takes a character array containing a string as an argument and prints the string backward. [Hint: Use String method toCharArray, which takes no arguments, to get a char array containing the characters in the String.]

**18.18 (Find the Minimum Value in an Array)** Write a recursive method recursiveMinimum that determines the smallest element in an array of integers. The method should return when it receives an array of one element.

**18.19 (Fractals)** Repeat the fractal pattern in [Section 18.9](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec9_html#ch18lev1sec9) to form a star. Begin with five lines (see [Fig. 18.21](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18fig21)) instead of one, where each line is a different arm of the star. Apply the “Lo feather fractal” pattern to each arm of the star.

**Fig. 18.21** | Sample outputs for [Exercise 18.19](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18que19).

**18.20 (Maze Traversal Using Recursive Backtracking)** The grid of #s and dots (.) in [Fig. 18.22](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18fig22) is a two-dimensional array representation of a maze. The #s represent the walls of the maze, and the dots represent locations in the possible paths through the maze. A move can be made only to a location in the array that contains a dot.

**Fig. 18.22** | Two-dimensional array representation of a maze.

Write a recursive method (mazeTraversal) to walk through mazes like the one in [Fig. 18.22](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18fig22). The method should receive as arguments a 12-by-12 character array representing the maze and the current location in the maze (the first time this method is called, the current location should be the entry point of the maze). As mazeTraversal attempts to locate the exit, it should place the character x in each square in the path. There’s a simple algorithm for walking through a maze that guarantees finding the exit (assuming there’s an exit). If there’s no exit, you’ll arrive at the starting location again. The algorithm is as follows: From the current location in the maze, try to move one space in any of the possible directions (down, right, up or left). If it’s possible to move in at least one direction, call mazeTraversal recursively, passing the new spot on the maze as the current spot. If it’s not possible to go in any direction, “back up” to a previous location in the maze and try a new direction for that location (this is an example of recursive backtracking). Program the method to display the maze after each move so the user can watch as the maze is solved. The final output of the maze should display only the path needed to solve the maze—if going in a particular direction results in a dead end, the x’s going in that direction should not be displayed. [Hint: To display only the final path, it may be helpful to mark off spots that result in a dead end with another character (such as '0').]

**18.21 (Generating Mazes Randomly)** Write a method mazeGenerator that takes as an argument a two-dimensional 12-by-12 character array and randomly produces a maze. The method should also provide the starting and ending locations of the maze. Test your method mazeTraversal from [Exercise 18.20](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18que20), using several randomly generated mazes.

**18.22 (Mazes of Any Size)** Generalize methods mazeTraversal and mazeGenerator of [Exercise 18.20](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18que20) and [Exercise 18.21](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec15_html#ch18que21) to process mazes of any width and height.

**18.23 (Time to Calculate Fibonacci Numbers)** Enhance the Fibonacci program of [Fig. 18.5](http://proquest.safaribooksonline.com/9780133813036/ch18lev1sec5_html#ch18fig05) so that it calculates the approximate amount of time required to perform the calculation and the number of calls made to the recursive method. For this purpose, callstatic System method currentTimeMillis, which takes no arguments and returns the computer’s current time in milliseconds. Call this method twice—once before and once after the call to fibonacci. Save each value and calculate the difference in the times to determine how many milliseconds were required to perform the calculation. Then, add a variable to the FibonacciCalculator class, and use this variable to determine the number of calls made to method fibonacci. Display your results.