**Chapter XIX**

**Recursion**

**Chapter XIX Topics**

19.1 Introduction

19.2 Pre-Recursion Assignment

19.3 Recursion Requires an Exit

19.4 Recursion Fundamentals

19.5 Recursive Return Methods

19.6 Fibonacci, a Recursive Nono

19.7 Evaluating Recursive Methods

19.8 Manipulating Parameters of Recursive Methods

19.9 Multiple Recursive Calls and the Tower of Hanoi

19.10 Why Recursion?

19.11 Summary

**19.1 Introduction**

You probably know very little about recursion. In English or in computer science. It is not the type of vocabulary that is common and you have already learned during this course that computer science has many vocabulary words that are quite unusual, but that really is true for any profession. It seems that in computer science as in other technical areas it somehow makes sense to ignore simple communication in favor of more educated language. Why say looping or repeating when you can say iterating. I would personally prefer the simpler words, but then students would not be prepared to the future academic world.

Recursion is a peculiar kind of control structure. Recursion performs an iterative process, or at least has the same result as an iterative control structure, without using **for**, **while** or **do...while**. Recursion is a special process, whereby a method calls itself to simulate iteration.

|  |
| --- |
| **Recursion Definition** |
| Recursion is the computer programming process, whereby a method calls itself. |

Recursion is somewhat of a bizarre topic. You will get a pretty good introduction to recursion during the first year course, but for the real practical reasons to use recursion, you need to wait for the second-year computer science course. Many processes are performed by a computer system recursively. Understanding a computer system thoroughly requires an understanding of recursion. Advanced computer algorithms make liberal use of recursion. Examples of such algorithms would serve little purpose here, but in the future you will certainly need to use recursion.

In other words, your teachers and I are doing you a favor. This topic could be skipped with the rationalization of not teaching something that is not used. Let some future teacher wrestle with a topic that really is not used very much right now. But that is not a student friendly approach. The title of this book is *Exposure Java*, which means that I believe in making the learning process manageable by repeated exposure. It is precisely the introduction of an advanced topic now that will make the learning of such a topic easier at a later date. And for those who plan to take the AP Examination, you will be pleased to know that recursion is very much part of the required knowledge.

|  |
| --- |
| **APCS Examination Alert** |
| Recursion is an important topic with frequent questions on both the APCS Examination.  The APCS Examination requires that a student can understand and evaluate recursive methods.  The APCS Examination does not require that students generate free response recursive solutions. |

**19.2 Pre-Recursion Assignment**

This chapter will start in an unusual manner. A free response problem will be explained shortly, which is similar to a problem presented to students at one of the earlier AP Computer science Examinations. The solution of this problem, called **Grid**, is done surprisingly easy with the concepts explained in this chapter. In an effort to prove a point you are presented with this problem before *recursion* is explained.

Try to solve the **Grid** problem without using recursion. In other words, write a solution to the problem using loop structures. Try to do this exercise first, before you learn about recursion, will help to motivate why recursion is such a useful, and powerful tool.

It is unlikely, but certainly possible, that you already understand recursion real well, and that you are able to write a recursive solution to the Grid problem. Doing so will defeat the purpose of this exercise. Play along and you will understand soon enough why it is important to try this problem iteratively first.

**The Grid Problem**

An image can be represented as a grid of black and white cells. Two cells in an image are part of the same object if each is black and there is a sequence of moves from one cell to the other, where each move is either horizontal or vertical to an adjacent black cell. For example, the diagram below represents an image that contains four separate objects, one of them consisting of a single cell.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Complete method **alterGrid** using the header below, such that all cells in the same object as **grid[row][col]** are set to **white**; otherwise **grid** is unchanged. **grid** is a two-dimensional array of **boolean** elements. An element value of **true** means a **black** cell and an element value of **false** means a **white** cell.

**public void alterGrid(boolean grid[ ][ ], int row, int col)**

The instructions for the Grid problem are very similar to the instructions that were given to students who took an early AP Computer Science Examination. No additional information, examples, hints, etc. will be given.

If you are a student at John Paul II High School, and Mr. Schram is in a good mood, you will receive extra credit for being able to write an *iterative*solution to the Grid problem. You will receive negative credit for turning in a *recursive*solution, which happens to be shown later on in this chapter, and proves that you do not follow instructions and know how to copy from a textbook.

Your solution needs to be printed, and it needs to be turned in before the first official lecture on the recursion chapter. Give this problem some serious thought. We will return to the grid at the end of the chapter, and then this exercise will make more sense.

**19.3 Recursion Requires an Exit**

The original *BASIC*programming language made frequent use of the **GOTO** control structure. **GOTO** is very much frowned upon today by computer scientists who do not like to see unconditional jumps in a program.

**Figure 19.1**

|  |
| --- |
| **100 REM Program that counts integers starting at 0**  **110 K = 0**  **120 K = K + 1**  **130 PRINT K**  **140 GOTO 120** |

More than once you have wondered about Mr. Schram’s sanity. This sudden switch to using BASIC is confirmation that Mr. Schram is not operating on all thrusters. We can certainly program in Java without using BASIC, but the concept is nicely demonstrated in BASIC, shown in figure 19.1. The variable **K** is incremented by one, displayed, and **GOTO** jumps back to line 120 to repeat the process. This process will go on for quite some time because nothing has been done to stop the loop. The **GOTO** statement specifies to go back to line 120 and repeat the process. There is no conditional statement that stops the repetition. A similar result can be achieved in Java with recursion, demonstrated by program **Java1901.java**, in figure 19.2.

## Figure 19.2

|  |
| --- |
| // Java1901.java  // This program demonstrates recursion without an exit.  // The recursive calls will continue until the program aborts  // with an error message.  public class Java1901  {    static int k = 0;    public static void main(String args[])  {  count();  }    public static void count()  {  k++;  System.out.print(k + " ");  count();  }    } |

**Figure 19.2 Continued**

|  |
| --- |
| **Java1901.java Output**  532 4533 4534 4535 4536 4537 4538 4539 4540 4541 4542 4543 4544 454  5 4546 4547 4548 4549 4550 4551 4552 4553 4554 4555 4556 4557 4558  4559 4560 4561 4562 4563 4564 4565 4566 4567 4568 4569 4570 4571 4  572 4573 4574 4575 4576 4577 4578 4579 4580 4581 4582 4583 4584 458  5 4586 4587 4588 4589 4590 4591 4592 4593 4594 4595 4596 4597 4598  4599 4600 4601 4602 4603 4604 4605 4606 4607 4608 4609 4610 4611 4  612 4613 4614 4615 4616 4617 4616 4619 4620 4621 4622 4623 4624 462  5 4626 4627 4628 4629 4630 4631 4632 4633 4634 4635 4636 4637 4638  4639 4640 4641 4642 4643 4644 4645 4646 4647 4648 4649 4650 4651 4  652 4653 4654 4655 4656 4657 4658 4659 4660 4661 4662 4663 4664 466  5 4666 4667 4668 4669 4670 4671 4672 4673 4674 4675 4676 4677 4678  4679 4680 4681 4682 4683 4684 4685 4686 4687 4688 4689 4690 4691 4  692 4693 4694 4695 4696 4697 4698 4699 4700 4701 4702 4703 4704 470  5 4706 4707 4708 4709 4710 4711 4712 4713 4714 4715 4716 4717 4716  4719 4720 4721 4722 4723 4724 4725 4726 4727 4728 4729 4730 4731 4  732 4733 4734 4735 4736 4737 4738 4739 4740 4741 4742 4743 4744 474  5 4746 4747 4748 4749 4750 4751 4752 4753 4754 4755 4756 4757 4758  4759 4760 4761 4762 4763 4764 4765 4766 4767 4768 4769 4770 4771 4  772 4773 4774 4775 4776 4777 4778 4779 4780 4781 4782 4783 4784 478  5 4786 4787 4788 4789 4790 4791 4792 4793 4794 4795 4796 4797 4798  4799 4800 4801 4802 4803 4804 4805 4806 4807 4808 4809 4810 4811 4  812 4813 4814 4815 4816 4817 4816 4819 4820 4821 4822 4823 4824 482  5 4826 4827 4828 4829 4830 4831 4832 4833 4834 4835 4836 4837 4838  4839 4840 4841 4842 4843 4844  This is followed by an **Illegal Operation** windows message. |

This program will count numbers, incrementing by 1, just like the BASIC program. It may seem that this odd kind of program will continue to count until the power is turned off or the computer is rebooted. Method **count** calls itself every time that **k** is incremented and displayed. There appears to be nothing that will stop this infinite counting process. But very soon the execution is halted with or without an error message. The problem is that you have a *stack overflow*.

For the moment being, do not be concerned about what exactly a *stack overflow error* is. Your program will crash with some type of problem. It may be sooner; it may be later; but accept the fact that any recursive process that is allowed to continue without a planned method of stopping the recursive calls, will **crash** the computer eventually. Absolutely guaranteed.

Recursion simulates iteration. With iterative control structures, whether you are using **for**, **while** or **do...while**, a condition is stated that stops the looping process. The same type of logic can be applied to any kind of recursive method. Prior to making a recursive call, some condition needs to be checked first. If this condition is true, then and only then, should the method call itself. Failure to provide an exit out of the recursive process will make the program crash. Some condition needs to be stated that checks the recursive calls.

One such example is demonstrated by **Java1902.java**, shown in figure 19.3. This program resembles the previous program with a minor modification. A small **if** condition has been added to stop the counting beyond 100. You will see two **count** methods, the first one uses *iteration* and the second one uses *recursion*. Many program examples, which follow in this chapter, will show two methods performing identical processes. The only difference is iteration and recursion.

## Figure 19.3

|  |
| --- |
| // Java1902.java  // This program demonstrates how to add an "exit" or "base case"  // to control recursive calls.  public class Java1902  {  static int k = 0;    public static void main(String args[])  {  System.out.println("CALLING ITERATIVE COUNT METHOD");  count1();  System.out.println("\n\nCALLING RECURSIVE COUNT METHOD");  count2();  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void count1()**  **{**  **for (int k = 1; k <= 100; k++)**  **System.out.print(k + " ");**  **}**    /\*\*\*\*\* RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void count2()**  **{**  **k++;**  **System.out.print(k + " ");**  **if (k < 100)**  **count2();**  **}**    } |

|  |
| --- |
| **Java1902.java Output**  CALLING ITERATIVE COUNT METHOD  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 16 19 20 21 22 2  3 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 4  3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6  3 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8  3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100  CALLING RECURSIVE COUNT METHOD  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 16 19 20 21 22 2  3 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 4  3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6  3 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8  3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100  EXECUTION TERMINATED |
| **Important Recursion Rule** |
| All recursive methods must have an *exit*that stops the  recursive process. The special case or condition that stops  the recursive calls is called the *base case*. |

Understanding the *base case*is the single most significant part of understanding recursion. Many examples will follow that demonstrate recursion, and with each example you will see that an *exit* or *base case*exists to control the execution.

**19.4 Recursion Fundamentals**

The first two program examples used for recursion demonstrations intentionally used methods without parameters. This is not very realistic, but it is simpler. In the previous section there were two goals. The first goal was to understand that recursion is a program process whereby a method calls itself. The second goal was to realize that all recursive methods need a special *base case*, which will stop the recursive calls.

These two concepts will be reinforced in this section. You will see a variety of common algorithms presented with two methods. Each program example will execute some process both iteratively and recursively. Right now you are more familiar with using loop structures for repetition. It may be easier to comprehend the recursive method when it is compared to a *more familiar* iterative method.

The use of parameters is extremely important in recursion. You will see that it is precisely the manipulation of parameters that allows recursion to simulate iteration so effectively. Now the best next step is to just examine many examples and discover what this strange recursion business is all about.

**The Skip Method**

The modest **skip** method is a nice, simple, yet practical method. You indicate how many lines you wish to skip and **skip** does the rest. Program **Java1903.java**, shown in figure 19.4, starts the parameter method examples rolling. The parameter **n**, in **skip**, dictates how many times **System.out.println();** will be used. A **for** loop is used with **n**, which controls how many times the loop will be iterated. Iterative **Skip** uses a loop that repeats the **println** statement, **n** times. Duplicating the iterative process with recursion means that **n** recursive calls need to be made. This suggests that we use an identical method heading for the recursive method.

The recursive **skip** method starts with protection against any values of **n** that are less than or equal to **0**. This condition also provides an exit for the method to stop the recursive calls.

If the condition **(n > 0)** is **true**, one line is skipped with **System.out.println();** Repetition is simulated by **skip** calling itself. The recursive call is simple enough, but the key factor is the parameter **n-1**. Each recursive call is made with a lesser value for **n**. Eventually **n** is reduced to the *base case*.

## Figure 19.4

|  |
| --- |
| // Java1903.java  // This program demonstrates the <Skip> method.  public class Java1903  {    public static void main(String args[])  {  System.out.println("CALLING ITERATIVE SKIP METHOD");  skip1(4);  System.out.println("CALLING RECURSIVE SKIP METHOD");  skip2(3);  System.out.println("EXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE SKIP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void skip1(int n)**  **{**  **for (int k = 1; k <= n; k++)**  **System.out.println();**  **}**    /\*\*\*\*\* RECURSIVE SKIP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void skip2(int n)**  **{**  **if ( n > 0)**  **{**  **System.out.println();**  **skip2(n-1);**  **}**  **}**  } |

**Figure 19.4 Continued**

|  |
| --- |
| **Java1903.java Output**  CALLING ITERATIVE SKIP METHOD  CALLING RECURSIVE SKIP METHOD  EXECUTION TERMINATED |

**The count(a,b) Method**

This chapter started by counting, but the counting was limited to starting at 1 and going on forever or stops at 100. You will now see program **Java1904.java**, in figure 19.5, with a more sophisticated **count** method. This method provides two parameters, **a** to indicate the starting number and **b** to indicate the last number.

## Figure 19.5

|  |
| --- |
| // Java1904.java  // This program demonstrates the <count(a,b)> method, which displays  // consecutive integers from a to b.  public class Java1904  {    public static void main(String args[])  {  System.out.println("CALLING ITERATIVE COUNT METHOD");  count1(10,25);  System.out.println("\n\nCALLING RECURSIVE COUNT METHOD");  count2(26,40);  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static void count1(int a, int b)  {  for (int k = a; k <= b; k++)  System.out.print(k + " ");  }    /\*\*\*\*\* RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void count2(int a, int b)**  **{**  **if (a <= b)**  **{**  **System.out.print(a + " ");**  **count2(a+1,b);**  **}**  **}**    } |

**Figure 19.5 Continued**

|  |
| --- |
| **Java1904.java Output**  CALLING ITERATIVE COUNT METHOD  10 11 12 13 14 15 16 17 16 19 20 21 22 23 24 25  CALLING RECURSIVE COUNT METHOD  26 27 28 29 30 31 32 33 34 35 36 37 38 39 40  EXECUTION TERMINATED |

Something very significant is shown with program **Java1904.java**. It is a nice example of manipulating parameters to simulate repetition. It also demonstrates the notion of a *base case* very nicely. Let us consider what is happening and look only at the recursive **count2** method, in figure 19.6, again.

**Figure 19.6**

|  |
| --- |
| public static void count2(int a, int b) // line 1  {  if (a <= b) // line 2  {  System.out.print(a + " "); // line 3  count2(a+1,b); // line 4  }  } |

**Line 1** shows the method heading. There is nothing unusual there. This can be the heading for an iterative or a recursive method. Two parameters are provided with **a** the first number to be displayed and **b** the last number shown.

**Line 2** is the all-important *base case*. The method displays integers from **a** to **b**. As long as it is true that **a <= b**, integers need to be displayed and the process needs to continue. If it is no longer true that **a <= b**, which means **a > b**, then the process must stop and the recursive calls stop.

**Line 3** is a simple output statement that displays the value of **a**, followed by two spaces to keep the numbers apart.

**Line 4** is the critical statement. This statement makes the recursive call, which start the process all over again. We do have a condition to stop making calls, but that condition is worthless if the value of **a** does not change. **a** must grow larger and eventually surpass **b**. This requirement is achieved with the **a+1** parameter. Each time that method **count2** is called, **a** increments by one.

## The next program example shows the count(a,b) yet one more time, but it is for the purpose of showing a very critical recursion concept. Is there a difference where the recursive call statement is placed? This may seem an odd question, but it is possible to pick from a variety of locations in a method to make a recursive call. In program Java1905.java, shown by figure 19.7, there are two count methods. This time both methods are recursive implementations. The difference is that count1 makes the recursive call after the number display print statement, and count2 makes the recursive call before the number display print statement.

## 

## Figure 19.7

|  |
| --- |
| // Java1905.java  // This program compares the difference between "post-recursive"  // calls and "pre-recursive" calls.  public class Java1905  {    public static void main(String args[])  {  System.out.println("CALLING POST-RECURSIVE COUNT METHOD");  count1(100,200);  System.out.println("\n\nCALLING PRE-RECURSIVE COUNT METHOD");  count2(100,200);  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* POST-RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void count1(int a, int b)**  **{**  **if (a <= b)**  **{**  **System.out.print(a + " ");**  **count1(a+1,b);**  **}**  **}**    /\*\*\*\*\* PRE-RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static void count2(int a, int b)**  **{**  **if (a <= b)**  **{**  **count2(a+1,b);**  **System.out.print(a + " ");**  **}**  **}**    } |

Method **count1** is exactly the same method that was shown earlier and I suspect that you see no surprises there. **count2** is another story. What will that display? It seems that the computer cannot get to the **System.out.print** statement, because every time there is an interruption and a recursive call is made before anything is displayed. Will any numbers be displayed?

**Figure 19.7 Continued**

|  |
| --- |
| **Java1905.java Output**  CALLING POST-RECURSIVE COUNT METHOD  100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115  116 117 116 119 120 121 122 123 124 125 126 127 128 129 130 131  132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147  148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163  164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179  160 161 162 163 164 165 166 167 168 169 190 191 192 193 194 195  196 197 198 199 200  CALLING PRE-RECURSIVE COUNT METHOD  200 199 198 197 196 195 194 193 192 191 190 169 168 167 166 165  164 163 162 161 160 179 178 177 176 175 174 173 172 171 170 169  168 167 166 165 164 163 162 161 160 159 158 157 156 155 154 153  152 151 150 149 148 147 146 145 144 143 142 141 140 139 138 137  136 135 134 133 132 131 130 129 128 127 126 125 124 123 122 121  120 119 116 117 116 115 114 113 112 111 110 109 108 107 106 105  104 103 102 101 100  EXECUTION TERMINATED |

Well plenty of numbers are displayed. The *post-recursive* method displays numbers in the expected sequence. The *pre-recursive* method displays all the numbers, but they are in reverse order. How does that happen? To answer that question we need to take a closer look at the manner that a computer processes the program statements. There exists something called a *stack*, which requires closer scrutiny and understanding.

**The Internal Stack and Recursion**

Java has a special internal data structure, called a **stack** that is hidden from the programmer. Programmers cannot use the stack directly, but you can detect its presence in a variety of ways. A **stack** is a data structure that will only access data from one end, usually called the **top**, in such a manner that it behaves like a **LIFO**. Data is accessed **L**ast **I**n, **F**irst **O**ut. Java uses this stack, lurking somewhere in the depths of the computer, to manage a variety of program execution features. How exactly this stack manages its business, and how the stack is implemented is of no concern to us. We simply accept the fact that the stack exists somewhere.

So what is the job of this mysterious stack? Consider the following. Every program statement is stored somewhere in computer memory, in machine code, understandable only to the computer and people who like to read binary machine code. Program execution sequence is handled by something called an **Index Pointer**. The **I**ndex **P**ointer, or **IP** for short, contains the next memory address of the line to be executed. In a *simple sequence* program without any method calls, consecutive program statements are usually placed in consecutive memory addresses. The value of the **IP** is increased by the memory size of the program statements. This great and not so complicated.

Everything is fine with this sequence of program statements. Sooner or later a method call is encountered. Now you know that a method can be called lots of times and the code is only written once, declared inside its very own class. This means that the executable code of a method sits somewhere in memory, and the regular program execution needs to jump to the method's location whenever the method is called. There still is no problem. Program execution cannot always follow consecutive memory addresses. Jumps sometimes need to be made at memory breaks and the Index Pointer sees no problem.

Methods present a different situation. When the method execution is finished, program execution needs to resume with the program statement following the method call. Simple, just make sure that a memory address is provided at the end of the method so that the Index Pointer gets a location to continue executing. Well it is not all that simple. The same method gets called from many different program locations. The start of the method is always at the same memory location. This makes life easy for the IP, but what address is used by the IP to resume execution? That can be different each time that the same method is called.

What comes to the rescue is our friendly neighborhood stack. The stack is a temporary storage structure that holds values in the unique LIFO manner. This system is custom-made to solve the *method-calling* problem. When a method is called, the memory location of the statement following the method call is added - called *pushed* - to the stack. The stack now continues to do service by storing local variable values of the method that are currently in scope. In other words, the stack takes care of the method we are executing. When the method is finished with its values, there will be a conspicuous memory location left in the stack. This value will find its way to the IP and program execution returns to the location where the method was called.

The stack really demonstrates its LIFO business when methods call other methods. Imagine that you start in the **main** method and call method **first**. Somewhere in the middle of **first**, method **second** is called. Now when **second** is finished where does it return? Should program execution continue back in the **main** method or method **first**? Hopefully you realize that execution returns to the location where the method was called. This means back to method **First**, and then after **first** is finished, execution continues in the main method. Do you see how neat the stack handle this business. First, a main method address is pushed on the stack, and then a **first** method address. The LIFO (Last In, First Out) controls the execution perfectly.

Digest this stack business, and return to methods **count1** and **count2,** in figure 19.7. Both methods are shown again in figure 19.8. In both methods the value of integer **a** is displayed. When either method is called, the local variables, which are **a** and **b**, are stored in the temporary stack, until the method is completely executed. Method **count1** finishes with the recursive call, which is why I call it *post-recursion* or *tail-recursion* in some places.

Method **count2** present a special situation. Before it is possible to display any numbers, a recursive call prevent completion of the method execution. This means that the local variable and their values will stay on the stack until the time comes to complete the execution. At the conclusion of the recursive calls, when it is no longer true that **(a <= b)**, which occurs when the *base case* is reached, the recursive call stop. Now there is lots of unfinished business and the stacks feeds all these unfinished calls back to the computer. What is the result? Everything executes backwards, because the stack behaves like a *Last-In-First-Out* data structure. Hence you see the very strange, but actually very reasonable, display of all the numbers in reverse order.

**Figure 19.8**

|  |
| --- |
| /\*\*\*\*\* POST-RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static void count1(int a, int b)  {  if (a <= b)  {  **System.out.print(a + " ");**  **count1(a+1,b);**  }  } |
| /\*\*\*\*\* PRE-RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static void count2(int a, int b)  {  if (a <= b)  {  **count2(a+1,b);**  **System.out.print(a + " ");**  }  } |

Program **Java1906.java**, in figure 19.9, will help to explain this stack business in greater detail. In this program, statements are inserted to alert you every time when information is pushed on the stack. Likewise, a message is displayed whenever information is popped off the stack.

**Figure 19.9**

|  |
| --- |
| // Java1906.java  // This program demonstrates how recursion can display output in reverse order.  public class Java1906  {    public static void main(String args[])  {  System.out.println("\n\nCALLING PRE-RECURSIVE COUNT METHOD");  count2(100,105);  System.out.println("\n\nEXECUTION TERMINATED");  }      /\*\*\*\*\* PRE-RECURSIVE COUNT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static void count2(int a, int b)  {  if (a <= b)  {  **System.out.println("Interrupting method completion; pushing " + a + " on stack.");**  **count2(a+1,b);**  **System.out.println("Returning to complete method; popping " + a + " from stack.");**  **System.out.println("Displaying popped value " + a);**  }  }    } |

|  |
| --- |
| **Java1906.java Output**  CALLING POST-RECURSIVE COUNT METHOD  Interrupting method completion; pushing 100 on stack.  Interrupting method completion; pushing 101 on stack.  Interrupting method completion; pushing 102 on stack.  Interrupting method completion; pushing 103 on stack.  Interrupting method completion; pushing 104 on stack.  Interrupting method completion; pushing 105 on stack.  Displaying popped value 105  Returning to complete method; popping 104 from stack.  Displaying popped value 104  Returning to complete method; popping 103 from stack.  Displaying popped value 103  Returning to complete method; popping 102 from stack.  Displaying popped value 102  Returning to complete method; popping 101 from stack.  Displaying popped value 101  Returning to complete method; popping 100 from stack.  Displaying popped value 100  EXECUTION TERMINATED |

|  |
| --- |
| **Fundamental Recursion Concepts** |
| All recursive methods require an *exit*  or *base case*  that stops the recursive process.  The *stack*controls recursion, since it controls the  execution sequence of methods, and stores local  method information.  Every method call requires the completion of the called  method, even if the execution sequence is interrupted,  by another recursive method call.  Incomplete recursive calls result in a *LIFO*execution  sequence. |

**19.5 Recursive Return Methods**

Every recursive method shown so far has been a **void** method. Is it preferable to use **void** methods for recursion? No, not at all, and many people will argue that **return** methods handles most recursive problems in a more natural and elegant manner. Watch out when experienced computer scientists speak of elegant solutions. There is a good chance that you will be totally lost. You look around and other people agree that this is *surely one of the more elegant implementations* they have seen. You decide to shut-up, after all, *intelligence of man in doubt till mouth opened*.

In the last section you saw that it is possible to *stack-up* incomplete methods and you will see that this is pretty much the norm for *return* methods. You will also see that *return* methods are frequently very brief, in comparison to the iterative solutions. If you understand recursion you may find that you need to write less program code.

**The sum Method**

Program **Java1907.java**, in 19.10, demonstrates method **sum**, which adds up all the integers from **1** to the provided parameter. This method is easy to write with a loop structure. You start with a temporary accumulator variable that is initialized to **0** and then add the loop control variable value each time through the loop. I suspect that you will have little difficulty understanding how the iterative **sum1** method works. Method **sum2** may be another story.

## Figure 19.10

|  |
| --- |
| // Java1907.java  // This program demonstrates the <sum> method.  // It also demonstrates recursion with return methods.  public class Java1907  {  public static void main(String args[])  {  System.out.println("CALLING ITERATIVE SUM METHOD");  System.out.println("1 + 2 + 3 + 4 + 5 + 6 = " + sum1(6));  System.out.println("\n\nCALLING RECURSIVE SUM METHOD");  System.out.println("1 + 2 + ... + 99 + 100 = " + sum2(100));  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE SUM \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int sum1(int n)**  **{**  **int sum = 0;**  **for (int k = 1; k <= n; k++)**  **sum += k;**  **return sum;**  **}**    /\*\*\*\*\* RECURSIVE SUM \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int sum2(int n)**  **{**  **if (n == 0)**  **return 0;**  **else**  **return n + sum2(n-1);**  **}**    } |

|  |
| --- |
| **Java1907.java Output**  CALLING ITERATIVE SUM METHOD  1 + 2 + 3 + 4 + 5 + 6 = 21  CALLING RECURSIVE SUM METHOD  1 + 2 + ... + 99 + 100 = 5050  EXECUTION TERMINATED |

**Figure 19.11**

|  |
| --- |
| **public static int sum2(int n) // Line 1**  **{**  **if (n == 0) // Line 2**  **return 0; // Line 3**  **else**  **return n + sum2(n-1); // Line 4**  **}** |

The output of the **sum** methods proves that the sum is computed correctly. You may wonder just how the *recursive sum* manages to do its job. Remember the recursion fundamentals which specified that unfinished business is stored on the temporary stack memory. So let us do another line-by-line analysis in figure 19.11, to decipher the logic of the recursive **sum2** method.

**Line 1** is the method heading. It shows that it must be a return method that will return an integer value. One parameter, **n**, is passed to the method to indicate the last value that must be added to the sum.

**Line 2** is the base case. If the value of **n == 0** then the recursive call stop.

**Line 3** returns **0**, if the base case is reached.

**Line 4** returns the current value of **n**, and then makes a recursive call to **sum2**, with the parameter reduced to **n-1**.

Perhaps line-by-line every statement makes sense, but you still do not see how this manages to add much of anything. Let me show you a specific example. Suppose that **Sum2** is called and **n == 4**. The trace that is shown in figure 19.12 is a method that will help you evaluate recursive methods. Later in this chapter you will see many examples of these method traces.

**Figure 19.12**

|  |  |  |
| --- | --- | --- |
| CALL # | n | Method **sum2** returns |
| 1  2  3  4  5 | 4  3  2  1  0 | 4 + Sum2(3)  3 + Sum2(2)  2 + Sum2(1)  1 + Sum2(0)  0 |
| **sum2(4) = 0 + 1 + 2 + 3 + 4 = 10** | | |

**The Factorial Method**

The *Factorial*method is a classic in computer science. Different algorithms may be seen in different computer science textbooks in a recursion chapter, but the factorial algorithm will be there. The logic of the recursive factorial algorithm is identical to **sum2** of the previous program. Unfinished business is stored on the stack until the base case is reached. When the base case finally stops the recursive calls, all the unfinished computations can be performed and the final answer is returned to the program statement, which called the method. Program **Java1908.java**, in figure 19.13, demonstrates the **fact** methods.

## Figure 19.13

|  |
| --- |
| // Java1908.java  // This program demonstrates the <fact> method.  public class Java1908  {  public static void main(String args[])  {  System.out.println("CALLING ITERATIVE FACTORIAL METHOD");  System.out.println("6 Factorial = " + fact1(6));  System.out.println("\n\nCALLING RECURSIVE FACTORIAL METHOD");  System.out.println("7 Factorial = " + fact2(7));  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE FACT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int fact1(int n)**  **{**  **int temp = 1;**  **for (int k = n; k > 0; k--)**  **temp \*= k;**  **return temp;**  **}**    /\*\*\*\*\* RECURSIVE FACT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int fact2(int n)**  **{**  **if (n == 0)**  **return 1;**  **else**  **return n \* fact2(n-1);**  **}**  } |

|  |
| --- |
| **Java1908.java Output**  CALLING ITERATIVE FACTORIAL METHOD  6 Factorial = 720  CALLING RECURSIVE FACTORIAL METHOD  7 Factorial = 5040  EXECUTION TERMINATED |

The factorial of a number **6**, usually denoted **6!**, is computed by multiplying

**6 x 5 x 4 x 3 x 2 x 1, which equals 720**

The iterative method sets a local variable, **temp**, to **1**. This value will be returned if the loop structure is not entered. If the loop is entered, **temp** is multiplied by the loop control variable from **n** down to **1**. The loop can count, and multiply in both directions, the backward counting was selected because it resembles the way that we compute factorial more closely.

The recursive solution needs to rely on our trusty stack to keep order of the values for each unfinished method call. **n** starts out as **6**, and makes repeated calls, before the base case **(n == 0)** is reached. Do not be surprised that **1** is returned at the base case. Mathematically it is true that **0!** equals **1**. Figure 19.14 shows the detailed trace of the recursive factorial computations.

**Figure 19.14**

|  |  |  |
| --- | --- | --- |
| CALL # | n | Method **fact2** returns |
| 1  2  3  4  5  6  7 | 6  5  4  3  2  1  0 | 6 \* Fact2(5)  5 \* Fact2(4)  4 \* Fact2(3)  3 \* Fact2(2)  2 \* Fact2(1)  1 \* Fact2(0)  1 |
| **fact2(6) = 1 \* 1 \* 2 \* 3 \* 4 \* 5 \* 6 = 120** | | |

**The Greatest Common Factor (gcf) Method**

Euclid’s algorithm for finding the **GCF** pops up at many computer science locations, or at least it pops up a lot in *Exposure Java*. I really like this algorithm and it demonstrates many concepts so nicely. Right now it will do an excellent job demonstrating how parameter manipulation makes recursion both possible and elegant. This word came up earlier, so what does elegant mean anyway? You thought that elegant applies to evening gowns? Think again, in computer science a solution can also be elegant. I am not sure that I can define elegant properly, but it appears that elegant solutions are very brief, straight forward and usually not the first thing coming to the mind of an unsuspecting computer science student. However, an elegant solution is frequently the most natural and *obvious* solution to many computer science veterans. First, let us review the iterative **gcf** very rapidly. Remember the algorithm? Refresh yourself quickly with this example shown in figure 19.15.

**Figure 19.15**

|  |
| --- |
| Find the **gcf** of **n1** (111) and **n2** (74).  Divide **n1** (111) by **n2** (74) and compute the remainder, which is **37**.  The remainder is not zero; you are not finished.  **n1** becomes **n2**  **n2** becomes the remainder  Divide **n1** (74) by **n2** (37) and compute the remainder, which is **0**.  Since the remainder is **0**, you are finished, and the **gcf** is **37**. |

Program **Java1909.java**, in figure 19.16, demonstrates these precise same steps in the iterative method of **gcf1**.If elegant means shorter then you cannot argue that the recursive solution is more elegant than the iterative variety. There are strong similarities in both solutions. In each case, the **rem**ainder of **n1** and **n2** is computed first. Also, in both cases a conditional statement checks if the remainder is equal to zero, and if it is ... the value of **n2** is returned. Check out the entire program first, and then we will return to examine the recursive method for the **gcf** in greater detail.

## Figure 19.16

|  |
| --- |
| // Java1909.java  // This program demonstrates the <gcf> method.  public class Java1909  {    public static void main(String args[])  {  System.out.println("CALLING ITERATIVE GCF METHOD");  System.out.println("GCF of 120 and 108 is " + gcf1(120,108));  System.out.println("\n\nCALLING RECURSIVE GCF METHOD");  System.out.println("GCF of 360 and 200 is " + gcf2(360,200));  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE GCF \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int gcf1(int n1, int n2)**  **{**  **int temp = 0;**  **int rem = 0;**  **do**  **{**  **rem = n1 % n2;**  **if (rem == 0)**  **temp = n2;**  **else**  **{**  **n1 = n2;**  **n2 = rem;**  **}**  **}**  **while (rem != 0);**  **return temp;**  **}**    /\*\*\*\*\* RECURSIVE GCF \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static int gcf2(int n1, int n2)  {  **int rem = n1 % n2;**  **if (rem == 0)**  **return n2;**  **else**  **return gcf2(n2,rem);**  **}**  } |

|  |
| --- |
| **Java1909.java Output**  CALLING ITERATIVE GCF METHOD  GCF of 120 and 108 is 12  CALLING RECURSIVE GCF METHOD  GCF of 360 and 200 is 40  EXECUTION TERMINATED |

Now the logic of the iterative and recursive solution is identical. Both methods use Euclid's algorithm. The recursive solution calls **cgf** when the base case is not satisfied. Keep in mind that it is not sufficient to repeat the process. The algorithm requires that **n1** becomes **n2** and **rem**ainder becomes **n2**.

Look how we can cleverly manipulate the parameters. The recursive method call, **return gcf2(n2,rem)**,passes the value of parameter **n2**, in the method call, to parameter **n1**, in the method heading. Also the value of parameter **rem** is passed to parameter **n2**.

**19.6 Fibonacci, a Recursive Nono**

The *Fibonacci Sequence* is a sequence of numbers that is formed by the sum of the two previous numbers in the sequence. The sequence starts with two **ones**, and from that point on, each number is formed by adding the two previous numbers. The first 14 Fibonacci numbers are shown below, followed by program **Java1910.java**, in figure 19.17, which displays two identical numbers for the same 10th Fibonacci number.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| **1** | **1** | **2** | **3** | **5** | **8** | **13** | **21** | **34** | **55** | **89** | **144** | **233** | **377** |

## Figure 19.17

|  |
| --- |
| // Java1910.java  // This program demonstrates the <fibo> method.  // Note that this style of recursion becomes unacceptably slow when the requested  // Fibonacci number becomes larger.  import java.util.\*;  public class Java1910  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("Enter requested Fibonacci number ===>> ");  int number = input.nextInt();  System.out.println("\n\nCALLING ITERATIVE FIBO METHOD");  System.out.println("Fibonacci(" + number + ") is " + fibo1(number));  System.out.println("\n\nCALLING RECURSIVE FIBO METHOD");  System.out.println("Fibonacci(" + number + ") is " + fibo2(number));  System.out.println("\n\nEXECUTION TERMINATED");  }    /\*\*\*\*\* ITERATIVE FIBO \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int fibo1(int n)**  **{**  **int n1 = 1;**  **int n2 = 1;**  **int n3 = 1;**  **if (n == 1 || n == 2)**  **return 1;**  **else**  **{**  **for (int k = 3; k <= n; k++)**  **{**  **n3 = n1 + n2;**  **n1 = n2;**  **n2 = n3;**  **}**  **}**  **return n3;**  **}**    /\*\*\*\*\* RECURSIVE FIBO \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int fibo2(int n)**  **{**  **if (n == 1 || n == 2)**  **return 1;**  **else**  **return fibo2(n-1) + fibo2(n-2);**  **}**  } |

|  |
| --- |
| **Java1910.java Output**  Enter requested Fibonacci number ===>> 10  CALLING ITERATIVE FIBO METHOD  The 6th Fibonacci number is 55  CALLING RECURSIVE FIBO METHOD  The 8th Fibonacci number is 55  EXECUTION TERMINATED |

The program, like others in this chapter, is shown with both the iterative and recursive methods. Does the logic of the method make sense? The recursive solution certainly appears to be much shorter. Now go ahead and test this program. You may realize why this section is titled, *Fibonacci, a Recursive Nono*. When you test the program, start by entering **10**, and then keep executing while you increase the requested number to 15, 20, 25, etc. The speed of your computer will make a difference, but in the neighborhood of the 30th or later number, you will observe that the recursive method demonstrates a very noticeable delay. You do not observe any delay? Well, you did not go far enough. I will guarantee you that at some point, the iterative solution will continue to appear instantaneous, while the recursive method has a serious efficiency problem.

Many students, who are new to recursion, mistakenly expect recursion to be faster than iteration. I assume that this is based on the shorter code. Look at any example so far and the recursive code is shorter than the iterative code. Perhaps it seems that less code means less execution. Well you got fooled. The nature of program execution efficiency is more complex.

First consider the general nature of recursion. It involves making multiple method calls to simulate the looping process. Multiple method calls each require a jump to another memory location and a large number of method calls causes a penalty in execution time. Now this is the general explanation and it accounts for a difference that is hardly noticeable. It does not explain the Fibonacci delay that can be observed even with the fastest computers on the market. Something else is going on. Look at the recursive statement, in figure 19.18, of the Fibonacci method. Do you see anything that you did not notice in any of the previous examples? Do not say that the variables have different names. Look at the nature of the recursive statements. Is there something odd, unusual that catches your eye? Perhaps you do not get the point, so look at a set of other recursive statements from two previous methods, and compare them. Is it clearer now?

**Figure 19.18**

|  |
| --- |
| **return fibo(n-1) + fibo(n-2);** |
| **return n \* fact(n-1);** |
| **return gcf(n2,rem);** |

Yes, you do see it, and you did not act like you did. The Fibonacci method generates two recursive calls in one statement. There is only one call for any of the other methods that we checked. Is this then bad? For small numbers this is not bad, but the consequences of these double recursive calls are disastrous for a large number. Each recursive statement doubles the number of recursive calls. Figure 19.19 demonstrates calling the 9th number. Check and see how many calls this ends up generating.

**Figure 19.19**

9

7 8

5 6 6 7

3 4 4 5 4 5 5 6

1 2 2 3 2 3 3 4 2 3 3 4 3 4 4 5

1 2 1 2 1 2 2 3 1 2 1 2 2 3 1 2 2 3 2 3 3 4

1 2 1 2 1 2 12 12 2 3

1 2

The diagram is only for the **9th** Fibonacci number. You see the large number of recursive calls, and you can also see how many times the same computation is performed. The efficiency of the recursive Fibonacci algorithm is horrible. Each higher number doubles the execution time. If the **30th** number takes **1** second, the **31st** takes **2** seconds, the **32nd** takes **4** seconds and the **50th** number will take **2,097,152** seconds. The pyramid diagram demonstrates how each level doubles the recursive calls of the previous level. Near the bottom of the pyramid, the number of calls gets a little raggedy, but that has to do with the fact that **fibo(1)** and **fibo(2)** bothprovide an **exit**. The main point with the Fibonacci example is that double recursive calls can rapidly yield an unacceptably time efficiency.

|  |
| --- |
| **Recursion Warning** |
| Avoid recursive method that make a single program  statement with multiple recursive calls, like the Fibonacci  sequence.  **return fibo(N-1) + fibo(N-2);** |

**19.7 Evaluating Recursive Methods**

You will also need to be able to *evaluate*recursive program modules. Given some method, can you determine what the value or values the method return when recursion is involved? This section gives you a variety of exercises to help you gain practice and confidence in evaluating recursive method. The tables are in the same format as those trace tables that were shown, earlier in this chapter.

|  |
| --- |
| **APCS Examination Alert** |
| Evaluating recursive method, like the examples that  follow will be required for both the multiple choice part and the free response part of the examination.  Few students are comfortable with this type of problem at the  first introduction. Most students achieve excellent evaluation  skills with repeated practice. |

**Exercise 01** Determine that **m1(5) == 25.**

|  |  |  |
| --- | --- | --- |
| **public int m1(int n)**  **{**  **if (n == 1)**  **return 25;**  **else**  **return m1(n-1);**  **}** | | |
| CALL # | n | Method m1 returns |
| 1  2  3  4  5 | 5  4  3  2  1 | m1(4)  m1(3)  m1(2)  m1(1)  25 |
| **m1(5) == 25** | | |

**Exercise 02** Determine that **m2(5)** causes anerror.

|  |  |  |
| --- | --- | --- |
| **public int m2(int n)**  **{**  **if (n == 1)**  **return 25;**  **else**  **return m2(n+1);**  **}** | | |
| CALL # | n | Method m2 returns |
| 1  2  3  4 | 5  6  7  8 | m2(6)  m2(7)  m2(8)  m2(9) |
| **Error results because the *base case* cannot be reached.** | | |

**Exercise 03**

Determine that **m3(5)** == **39**

|  |  |  |
| --- | --- | --- |
| **public int m3(int n)**  **{**  **if (n == 1)**  **return 25;**  **else**  **return n + m3(n-1);**  **}** | | |
| CALL # | n | Method m3 returns |
| 1  2  3  4  5 | 5  4  3  2  1 | 5 + m3(4)  4 + m3(3)  3 + m3(2)  2 + m3(1)  25 |
| **m3(5) == 25 + 2 + 3 + 4 + 5 == 39** | | |

**Exercise 04** Determine that **m4(1)** == **10**

|  |  |  |
| --- | --- | --- |
| **public int m4(int n)**  **{**  **if (n == 5)**  **return 0;**  **else**  **return n + m4(n+1);**  **}** | | |
| CALL # | n | Method m4 returns |
| 1  2  3  4  5 | 1  2  3  4  5 | 1 + m4(2)  2 + m4(3)  3 + m4(4)  4 + m4(5)  0 |
| **m4(1) == 0 + 4 + 3 + 2 + 1 = = 10** | | |

**Exercise 05** Determine that **m5(6)** == **38**

|  |  |  |
| --- | --- | --- |
| **public int m5(int n)**  **{**  **if (n == 1 || n == 0)**  **return 0;**  **else**  **return n + m5(n-1) + m5(n-2);**  **}** | | |
| CALL # | n | Method m5 returns |
| 1  2  3  4  5  6 | 6  5  4  3  2  1 | 6 + m5(5) + m5(4)  5 + m5(4) + m5(3)  4 + m5(3) + m5(2)  3 + m5(2) + m5(1)  2 + m5(1) + m5(0)  0 |
| **m5(6) == 0 + 2 + 3 + 4 + 5 + 6 == 20** | | |

Using previous techniques, the answer appears to be **20**. However, you are expected to determine that **m5(6) == 38**. Something is wrong, and the problem is that Exercise 05 involves nasty double recursive calls. The tree structure below determines the value that must be added at each method call. A call to **m5(1)** or **m5(0)** will result in **0** being added.

**6**

**5 4**

**4 3 3 2**

**3 2 2 1 2 1 1 0**

**2 1 1 0 1 0 1 0**

**Now we have 6 + 5 + 4 + 4 + 3 + 3 + 2 + 3 + 2 + 2 + 2 + 2 == 38**

**Exercise 06** Determine that **m6(5)** == **120**

|  |  |  |
| --- | --- | --- |
| **public int m6(int n)**  **{**  **if (n == 1)**  **return 1;**  **else**  **return n \* m6(n-1);**  **}** | | |
| CALL # | n | Method m6 returns |
| 1  2  3  4  5 | 5  4  3  2  1 | 5 \* m6(4)  4 \* m6(3)  3 \* m6(2)  2 \* m6(1)  1 |
| **m6(5) == 1 \* 2 \* 3 \* 4 \* 5 == 120** | | |

**Exercise 07** Determine that **m7(5,6)** == **30**

|  |  |  |  |
| --- | --- | --- | --- |
| **public int m7(int a, int b)**  **{**  **if (a == 0)**  **return 0;**  **else**  **return b + m7(a-1,b);**  **}** | | | |
| CALL # | a | b | Method m7 returns |
| 1  2  3  4  5  6 | 5  4  3  2  1  0 | 6  6  6  6  6  6 | 6 + m7(4,6)  6 + m7(3,6)  6 + m7(2,6)  6 + m7(1,6)  6 + m7(0,6)  0 |
| **m7(5,6) = 0 + 6 + 6 + 6 + 6 + 6 = 30** | | | |

**Exercise 08** Determine that **m8(4,3)** == **81**

|  |  |  |  |
| --- | --- | --- | --- |
| **public int m8(int a, int b)**  **{**  **if (a == 0)**  **return 1;**  **else**  **return b \* m8(a-1,b);**  **}** | | | |
| CALL # | a | b | Method m8 returns |
| 1  2  3  4  5 | 4  3  2  1  0 | 3  3  3  3  3 | 3 \* m8(3,3)  3 \* m8(2,3)  3 \* m8(1,3)  3 \* m8(0,3)  1 |
| **m8(4,3) == 1 \* 3 \* 3 \* 3 \* 3 == 81** | | | |

**Exercise 09** Determine that **m9(4,3)** == **64**

|  |  |  |  |
| --- | --- | --- | --- |
| **public int m9(int a, int b)**  **{**  **if (b == 0)**  **return 1;**  **else**  **return a \* m9(a,b-1);**  **}** | | | |
| CALL # | a | b | Method m9 returns |
| 1  2  3  4 | 4  4  4  4 | 3  2  1  0 | 4 \* m9(4,2)  4 \* m9(4,1)  4 \* m9(4,0)  1 |
| **m9(4,3) == 1 \* 4 \* 4 \* 4 == 64** | | | |

**Exercise 10** Determine that **m10(7,3) == 17**

|  |  |  |  |
| --- | --- | --- | --- |
| **public int m10(int a, int b)**  **{**  **if (a < b)**  **return 5;**  **else**  **return b + m10(a-1,b+1);**  **}** | | | |
| CALL # | a | b | Method m10 returns |
| 1  2  3  4 | 7  6  5  4 | 3  4  5  6 | 3 \* m10(6,4)  4 \* m10(5,5)  5 \* m10(4,6)  5 |
| **m10(7,3) == 5 + 5 + 4 + 3 == 17** | | | |

**19.8 Manipulating Parameters of**

**Recursive Method**

We are now going to move recursion up one notch on the difficulty ladder. All the previous examples have been fairly straight forward and the comparisons between iterative and recursive solutions always involved the same parameters. It did not really seem to matter how the method heading was written.

In the past, recursion and iteration made a difference in the body of the method, not the heading. This is now going to change and in this section you will see that recursion may very well require that the parameter lists in the method headings are altered to accommodate the recursion requirement. We shall use the *Linear Search*and *Binary Search*algorithmsto demonstrate this concept.

**The LinearSearch Method**

First, we need to review the *linearSearch*method, implemented iteratively. This implementation of the search algorithm returns the index (**k**) of the search item (**key**)**,** or it returns **k == -1** if the search is not successful. The recursive implementation needs to perform the same process, just done recursively. Both methods are shown by program **Java1911.java**, in figure 19.20.

## Figure 19.20

|  |
| --- |
| // Java1911.java  // This program demonstrates the <linear> method, which performs  // a Linear Search of a sorted array.  public class Java1911  {  public static void main(String args[])  {  int list[] = new int[100];  assign(list);  display(list);  System.out.println("\n\nCALLING ITERATIVE LINEAR SEARCH METHOD");  System.out.println("Number 425 is located at index " + linear1(list,425));  System.out.println("Number 211 is located at index " + linear1(list,211));  System.out.println("\n\nCALLING RECURSIVE LINEAR SEARCH METHOD");  System.out.println("Number 375 is located at index " + linear2(list,375,0));  System.out.println("Number 533 is located at index " + linear2(list,533,0));  System.out.println("\n\nEXECUTION TERMINATED");  }    public static void assign(int list[])  {  for (int k = 0; k < list.length; k++)  list[k] = 100 + k \* 5;  }    public static void display(int list[])  {  for (int k = 0; k < list.length; k++)  System.out.print(list[k] + " ");  }    /\*\*\*\*\* ITERATIVE LINEAR SEARCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int linear1(int list[], int key)**  **{**  **boolean found = false;**  **int k = 0;**  **while (k < list.length && !found)**  **{**  **if (list[k] == key)**  **found = true;**  **else**  **k++;**  **}**  **if (found)**  **return k;**  **else**  **return -1;**  **}**    /\*\*\*\*\* RECURSIVE LINEAR SEARCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int linear2(int list[], int key, int k)**  **{**  **if (k == list.length)**  **return -1;**  **else**  **if (list[k] == key)**  **return k;**  **else**  **return linear2(list,key,k+1);**  **}**  } |

|  |
| --- |
| **Java1911.java Output**  100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175  160 165 190 195 200 205 210 215 220 225 230 235 240 245 250 255  260 265 270 275 280 285 290 295 300 305 310 315 320 325 330 335  340 345 350 355 360 365 370 375 380 385 390 395 400 405 410 415  420 425 430 435 440 445 450 455 460 465 470 475 480 485 490 495  500 505 510 515 520 525 530 535 540 545 550 555 560 565 570 575  580 585 590 595  CALLING ITERATIVE LINEAR SEARCH METHOD  Number 425 is located at index 65  Number 211 is located at index -1  CALLING RECURSIVE LINEAR SEARCH METHOD  Number 375 is located at index 55  Number 533 is located at index -1  EXECUTION TERMINATED |

You will not get an explanation of the **linearSearch** algorithm. If you have forgotten, both of the search algorithms are explained in the previous chapter. Only the heading of the iterative **linearSearch** method is shown in figure 19.21, for direct comparison with the recursive implementation. The recursive solution has one extra parameter, **k**, which is not seen in the iterative method.

**Figure 19.21**

|  |
| --- |
| /\*\*\*\*\* ITERATIVE LINEAR SEARCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int linear1(int list[], int key)** |
| /\*\*\*\*\* RECURSIVE LINEAR SEARCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int linear2(int list[], int key, int k)**  **{**  **if (k== list.length)**  **return -1;**  **else**  **if (list[k] == key)**  **return k;**  **else**  **return linear2(list,key,k+1);**  **}** |

So why is there a need to alter the method heading for a recursive method? The logic is with the *loop control variable* **k**. The variable **k** is incremented throughout the **while** loop. When you change the loop structure and implement recursion, the parameters **list** and **key** are still needed.

And where exactly is our friend **k** located? Using the same logic as the iterative method cannot work because **k** is defined locally, and initialized to **0**. This works fine for a loop structure, but a recursive solution would constantly reset **k** to **0**. The trick is that **k** needs to become a parameter.

The addition of the **k** parameter solves our problem very nicely. Each time that a recursive call is made, the value of **k+1** is passed to the next **linearSearch** heading, parameter **k**. This will achieve the incrementing of the array index.

Since **k** is now a parameter, you will notice that the **linearSearch** method does not assign any initial value to **k**. This is not possible, since it would constantly restart the index value. **k** still needs an initial value, and that is accomplished with the first call to **linearSearch** done in the program’s main method. One parameter needs to be added in the method call to match **k**, and the constant value **0** works very nicely. **k**  starts as **0** and is now ready to search the list.

**The Binary Search Method**

The lessons that you learned with the recursive **linearSearch** method will help in understanding the **binarySearch** method. **linearSearch** only required that one parameter, **k**, is added to the method heading. You will find that **binarySearch** is going to be a little more complicated. But first review the logic of the *Binary Search* algorithm by looking at program **Java1912.java**, in figure 19.22.

## Figure 19.22

|  |
| --- |
| // Java1912.java  // This program demonstrates the <binary> method, which performs  // a Binary Search of a sorted array.  public class Java1912  {  public static void main(String args[])  {  int list[] = new int[100];  assign(list);  display(list);    System.out.println("\n\nCALLING ITERATIVE BINARY SEARCH METHOD");  System.out.println("Number 425 is at index " + binary1(list,425));  System.out.println("Number 211 is at index " + binary1(list,211));  System.out.println("\n\nCALLING RECURSIVE BINARY SEARCH METHOD");  System.out.println("Number 375 is at index " + binary2(list,375,0,99));  System.out.println("Number 533 is at index " + binary2(list,533,0,99));  System.out.println("\n\nEXECUTION TERMINATED");  }    public static void assign(int list[])  {  for (int k = 0; k < list.length; k++)  list[k] = 100 + k \* 5;  }    public static void display(int list[])  {  for (int k = 0; k < list.length; k++)  System.out.print(list[k] + " ");  }    /\*\*\*\*\* ITERATIVE BINARY SEARCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int binary1(int list[], int key)**  **{**  **int lo = 0;**  **int hi = list.length-1;**  **int mid = 0;**  **boolean found = false;**  **while (lo <= hi && !found)**  **{**  **mid = (lo + hi) /2;**  **if (list[mid] == key)**  **found = true;**  **else**  **if (key > list[mid])**  **lo = mid + 1;**  **else**  **hi = mid - 1;**  **}**  **if (found)**  **return mid;**  **else**  **return -1;**  **}**    /\*\*\*\*\* RECURSIVE BINARY SEARCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **public static int binary2(int list[], int key, int lo, int hi)**  **{**  **int mid = 0;**  **if (lo > hi)**  **return -1;**  **else**  **{**  **mid = (lo + hi) /2;**  **if (list[mid] == key)**  **return mid;**  **else**  **if (key > list[mid])**  **return binary2(list,key,mid+1,hi);**  **else**  **return binary2(list,key,lo,mid-1);**  **}**  **}**  } |

**Figure 19.22 Continued**

|  |
| --- |
| **Java1912.java Output**  100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175  160 165 190 195 200 205 210 215 220 225 230 235 240 245 250 255  260 265 270 275 280 285 290 295 300 305 310 315 320 325 330 335  340 345 350 355 360 365 370 375 380 385 390 395 400 405 410 415  420 425 430 435 440 445 450 455 460 465 470 475 480 485 490 495  500 505 510 515 520 525 530 535 540 545 550 555 560 565 570 575  580 585 590 595  CALLING ITERATIVE BINARY SEARCH METHOD  Number 425 is at index 65  Number 211 is at index -1  CALLING RECURSIVE BINARY SEARCH METHOD  Number 375 is at index 55  Number 533 is at index -1  EXECUTION TERMINATED |

The important feature of the *binary search*is the *search range*. In a *linear search* the search range is the entire array. In a *binary search* the initial search range is the entire array. The index of the first element of the first search is **0**, and that value is assigned to **lo**. The index of the last element of the search range is **n-1** (N equals array length), and that value is assigned to **hi**. The *binary search* performs a comparison with the element in the middle of the search range. Variable **mid** is computed as the mean of **lo** and **hi**. After each comparison, the search range is “chopped” in half. A new **lo** or a new **hi** must be determined if the desired item is not found. This process is repeated until the search item is found or **lo** becomes greater than **hi**.

The iterative solution defines **lo**, **hi** and **mid** locally. Recursively, there is no problem with using **mid** locally. The value of **mid** must be computed each time that a new location is searched. However, variables **lo** and **hi** are a different story. These index values change with each recursive call. An initial call, shown in figure 19.23, to **binarySearch**, made from the main method uses a method call with similar constant-value-logic like the **linearSearch** method.

**Figure 19.23**

|  |
| --- |
| **System.out.println("Number 375 is at index " + binary2(list,375,0,99));** |

If the search item, **key** is not found and **key** is greater than **list[mid]**, the lower half of the list can be ignored. This means that the **hi** value remains unchanged, but we have a new **lo**, which is one greater than **mid**. This will result in the recursive method call shown in figure 19.24.

**Figure 19.24**

|  |
| --- |
| **return binary2(list,key,mid+1,hi);** |

On the other hand, if **key** is less than **list[mid]**, the upper half of the list can be ignored. This means that the **lo** value remains unchanged, and we have a new **hi**, which is one less than **mid**. This will result in the recursive method call shown in figure 19.25.

**Figure 19.25**

|  |
| --- |
| **return binary2(list,Key,lo,mid-1);** |

Ponder over this question. In this section you were shown a *linear search* method and a *binary search* method that are implemented recursively by adding parameters to the method heading. Can you write these methods recursively with the same exact parameter list as the iterative solutions?

|  |
| --- |
| **Recursion and Parameters** |
| Changing existing iterative methods into recursive methods is an excellent exercise in obtaining proficiency with recursion.  Frequently it may be necessary to add parameters to the headings of the recursive methods.  Two examples of methods that require additional parameters in the recursive implementation are the *Linear Search*and the *Binary Search*. |

**19.9 Multiple Recursive Calls and**

**The Tower of Hanoi**

The *Tower of Hanoi* is a classic problem. This problem fits beautifully in a recursion chapter because of the elegant - *meaning trouble for the recursively challenged* - recursive solution to the problem and the lessons that are presented in understanding recursion.

So what is the problem? Look at the pictures below and on the next page. Three towers, or pegs, are shown. The towers are labeled *A*, *B*and *C*. Tower *A*has five numbered disks. The disks are stacked with the largest disk on the bottom. Your problem is to move all the disks from tower *A*to tower *C*. There is a catch, though. You can only move one disk at a time and you are not allowed to stack a larger disk on top of a smaller disk. With the three pegs, it is possible to accomplish this mission. The second peg can be used as an auxiliary peg.

**Tower of Hanoi at the Start of the 5-Disk Problem**

**4 4**

**1 1**

**2 2**

**3 3**

**5 5**

**Peg A Peg B Peg C**

**Tower of Hanoi at the End of the 5-Disk Problem**

**3 3**

**4 4**

**1 1**

**2 2**

**5 5**

**Peg A Peg B Peg C**

**The Trivial One-Disk Problem**

Let us look at a variety of problems, starting with the simplest possible situation. Suppose that you only need to move one disk from peg **A** to peg **C**. Would that be any kind of problem?

**1 1**

**Peg A Peg B Peg C**

How do we solve the **1-Disk** problem? In exactly one step that be described with the single statement:

**Move Disk1 from Peg A to Peg C**

**1 1**

**Peg A Peg B Peg C**

The solution to the *1-Disk*problem is commonly known as the *trivial*case. The example was shown here not to insult your intelligence, but to make sure that you focus on the *trivial*case. We will be seeking a recursive solution to this problem and it may just turn out that the *trivial*case also is very much part of a solution with more than one disk.

Ironically, students have the biggest problem in properly setting up the base case of a recursive solution to a problem. Basically, this means to start by considering what is necessary *to get out*.

**The Easy Two-Disk Problem**

The *1-Disk*problem was too easy. Let us move up to the big time and solve the *2-Disk*version of the Tower of Hanoi. It will take three disk moves to move two disks from peg **A** to peg **C**.

**Step 1**

**1 1**

**2 2**

**Peg A Peg B Peg C**

**Step 2**

**1 1**

**2 2**

**Peg A Peg B Peg C**

**Step 3**

**1 1**

**2 2**

**Peg A Peg B Peg C**

**Step 4**

**1 1**

**2 2**

**Peg A Peg B Peg C**

The *2-Disk* situation is also pretty easy to follow. It can be summed up with the following three disk moves:

**Move Disk1 from Peg A to Peg B**

**Move Disk2 from Peg A to Peg C**

**Move Disk1 from Peg B to Peg C**

I will not bother you with all the pictures of a *3-Disk*solution. Study the steps and see if they make sense to you. If necessary, take some objects and simulate the *Tower of Hanoi* problem. Here is the *3-Disk*solution.

**Move Disk1 from Peg A to Peg C**

**Move Disk2 from Peg A to Peg B**

**Move Disk1 from Peg C to Peg B**

**Move Disk3 from Peg A to Peg C**

**Move Disk1 from Peg B to Peg A**

**Move Disk2 from Peg B to Peg C**

**Move Disk1 from Peg A to Peg C**

The complexity of this problem grows very rapidly as the number of disks grows. The number of moves that are made by the solution also grows very rapidly. The *1-Disk*problem takes *1*move. The *2-Disk*problem takes *3*moves. The *3-Disk*problem takes *7*moves. And a *4-Disk*problem takes *15*moves. Perhaps you have recognized the pattern by now. The number of moves is **2n-1**, where **n** is the number of disks that need to be moved.

|  |
| --- |
| **Tower of Hanoi Formula** |
| There is a pattern in the number of minimum moves  to solve a given *Tower of Hanoi* problem.  With **n** the number of disks to be moved, the formula  below computes the number or required moves.  **2n - 1** |

It may seem like a very difficult task to write a program that solves the *Tower of Hanoi* puzzle. It would not be so tricky for a fixed number of disks, like **4**, but you may be perplexed by writing a general program that will handle any given number of disks.

The *Tower of Hanoi* program in this chapter solves the puzzle, and furthermore it solves the puzzle with an absolute minimum of code. The code is recursive and it will not be easy to understand at first or second glance.

You may see statements in various computer science textbooks with a statement that *the correctness of the Tower of Hanoi program is easily established*. Such statements are intimidating to students who are confused by some bizarre looking programming code. Take your time to digest the logic and syntax. Listen to your teacher’s explanation and allow this recursive business to start soaking in. You will need to be pretty drenched before you feel comfortable. But suddenly, one day, it all starts to click and recursion will be your friend from then on.

Program **Java1913.java**, in figure 19.26, solves this interesting puzzle. Several sample executions follow, to demonstrate the output of the program. Some explanation will be given later about the logic of the program solution, but right now there is another objective. You will probably not fully comprehend this solution until it is revisited in the second year course. Take a look at the solution. Never mind whether it makes sense or not. Isn’t the method that moves the disks amazingly short?

## Figure 19.26

|  |
| --- |
| // Java1913.java  // This program solves the "Tower of Hanoi" puzzle.  import java.util.\*;  public class Java1913  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nHow many disks are in the tower ===>> ");  int disks = input.nextInt();  solveHanoi('A','B','C',disks);  System.out.println("\n\nEXECUTION TERMINATED");  }    public static void solveHanoi(char s, char t, char d, int n)  // s - source peg  // t - temporary peg  // d - destination peg  // n - number of disks  {  **if (n != 0)**  **{**  **solveHanoi(s,d,t,n-1);**  **System.out.println("Move Disk " + n + " From Peg " + s + " to Peg " + d);**  **solveHanoi(t,s,d,n-1);**  **}**  }  } |

|  |
| --- |
| **Java1913.java Output #1**  How many disks are in the tower ===>> 1  Move Disk 1 From Peg A to Peg C  EXECUTION TERMINATED |
| **Java1913.java Output #2**  How many disks are in the tower ===>> 2  Move Disk 1 From Peg A to Peg B  Move Disk 2 From Peg A to Peg C  Move Disk 1 From Peg B to Peg C  EXECUTION TERMINATED |
| **Java1913.java Output #3**  How many disks are in the tower ===>> 4  Move Disk 1 From Peg A to Peg B  Move Disk 2 From Peg A to Peg C  Move Disk 1 From Peg B to Peg C  Move Disk 3 From Peg A to Peg B  Move Disk 1 From Peg C to Peg A  Move Disk 2 From Peg C to Peg B  Move Disk 1 From Peg A to Peg B  Move Disk 4 From Peg A to Peg C  Move Disk 1 From Peg B to Peg C  Move Disk 2 From Peg B to Peg A  Move Disk 1 From Peg C to Peg A  Move Disk 3 From Peg B to Peg C  Move Disk 1 From Peg A to Peg B  Move Disk 2 From Peg A to Peg C  Move Disk 1 From Peg B to Peg C  EXECUTION TERMINATED |

If you do not think that the solution is short, get behind your computer and write a solution that does not use recursion. It can be done iteratively and your teacher will offer you some major extra credit if your iterative solution is shorter than the recursive solution.

You have noticed in the examples used in this chapter so far that recursive solutions have been shorter than iterative solutions. For the most part, this has fallen in the *whoop..tee..doo* category. The difference in code has not been significant enough to be overly concerned. With this program you cannot compare. Will an iterative solution be shown? No, it will not, since it is an extra credit assignment. However, you can probably still appreciate that this **solveHanoi** method is very short, and accomplished a lot.

How does this short method manage to do its job? The secret is in some double recursive calls that have not been used before. Do not get confused and think that this resembles the recursive Nono that you were warned about with the Fibonacci method. With Fibonacci, two recursive calls were made in one program statement. With **solveHanoi** you observe two separate program statements, and each one makes a recursive call.

So how do we make sense out of this? I will assume that you understand clearly how to move a single disk. Now this simplest problem does not require a loop or any type of recursion. This simplest situation not only occurs if you have a problem with one disk, it also occurs in all the problems when all the disks are on the destination peg, except for the last disk. This one-disk move shall be known as the *trivial case*.

Now recursion requires a general statement that solves the problem for the situation that is not the base case. For instance, with the **fact** method the general solution for integer **n** is to multiply **n \* fact(n-1)**, provided **n is greater than 0**. Can you think about the *Tower of Hanoi* in a general way? Do not think about Java program code right now. State the general steps required to solve this puzzle.

|  |
| --- |
| **Tower of Hanoi General Solution** |
| Move **n-1** disks from Source Peg to Auxiliary Peg  Move the **nth** disk from Source Peg to Destination Peg  Move **n-1** disks from Auxiliary Peg to Destination Peg |

These three English program solution statements are very significant. The middle statement sure sounds a bunch like the trivial case. Essentially, the statement says that after you get rid of **n-1** disks that are on top of the **nth** disk, the **nth** disk can now easily be moved to its final destination. With the **nth** disk in the correct place, you can now move **n-1** disks from the temporary peg to the final destination on top of the **nth** disk. These three steps are illustrated below.

**Move n-1 Disks from Peg A to Peg B**

**N-1 N-1**

**N N**

**1 1**

**2 2**

**N-2 N-2**

**Peg A Peg B Peg C**

**Move Nth Disk from Peg A to Peg C**

**N-1 N-1**

**N N**

**1 1**

**2 2**

**N-2 N-2**

**Peg A Peg B Peg C**

**Move N-1 Disks from Peg B to Peg C**

**N-1 N-1**

**N N**

**1 1**

**2 2**

**N-2 N-2**

**Peg A Peg B Peg C**

Moving **n-1** disks from one peg to another peg should sound like a similar solution required with one less disk. It also sounds like this process continues until only one disk is left. Those students whose *recursive-understanding-light-bulbs* have acquired a considerable glow, are smelling serious recursion here. But something else is going on. Every one of the recursive examples presented in this chapter only had a single statement to make recursive calls. The Fibonacci method did call two methods recursively, but it was in one statement.

Take a look at the **solveHanoi** method again, in figure 19.27. You will recognize the three English statements of the general solution in the Java code. You will also note that there are two separate statements that make recursive calls. Do not think that this is unusual. There are many problems in computer science that are solved with multiple recursive calls.

But there remains a significant problem to solve. If you simulate the *Tower of Hanoi* puzzle with some disks, blocks, books or any other objects, you will rapidly notice a pattern. The **source-peg**  and **destination-peg** are constantly changing. With three pegs the initial aim is to move **n** disks from *A*to *C*. But first you need to move **n-1** disks from *A*to *B*. This means that for **n-1** disks the destination peg is now *B*. And later when these same disks have to be moved, *B*will be the source peg.

This constantly changing of source and destination is the secret in the solution. The parameter list of the **solveHanoi** method shows four parameters. The first three parameters are used to store the peg designation with characters *A*, *B*or *C*. The fourth parameter, **n**, passes the number of disks to be moved.

**Figure 19.27**

|  |
| --- |
| public static void solveHanoi(char s, char t, char d, int n)  // s - source peg  // t - temporary peg  // d - destination peg  {  if (n != 0)  {  solveHanoi(s,d,t,n-1);  System.out.println("Move Disk " + n + " From Peg " + s + " to Peg " + d);  solveHanoi(t,s,d,n-1);  }  } |
| **METHOD CALL IN THE MAIN METHOD**  **solveHanoi('A','B','C',disks);** |

Notice how the parameters are called **s**(ource, **t**(emporaryand **d**(estination. The initial call, made in the main method, passes **'A'** to **s**, **'B'** to **t** and **'C'** to **d**. This all seems to match what is desired.

Now our first job is to move **n-1** disks from **source** to **temp**. Look at the first recursive statement. **source** is passed to **source**, **destination** is passed to **temp**, and **temp** is now passed to **destination**. The parameters of the recursive calls are constantly changing what is **source**, **temp** and **destination**.

Notice in the second recursive call that now **temp** is passed to **source**. This is precisely what is needed, because you have **n-1** disks sitting on the temporary peg. After the **nth** disk is moved to its proper location, it is time to move the **n-1** disks, and they are located on the temporary peg.

Some students are not convinced. You do buy the movement of the single **nth** disk, but there seems to be some casual movement of **n-1** disks going on. Is there not a rule about moving one disk at a time? You are totally correct and totally alert. After the **nth** disk is in place we ignore that disk and ask how we can move **n-1** disks? This is done by first moving **n-2** disks out of the way to a temporary location, so that the (**n-1th)** disk can find its home. This process is continued until the trivial case makes life simple by moving a single disk.

Did we forget to consider how the recursion will stop? No, it has not been forgotten. Method **solveHanoi** can only generate recursive calls, and move a disk whenever **(n != 0)** is **true**. Any time that the disk count becomes **0**, it is time to say good bye and return to complete unfinished business.

**19.10 Why Recursion?**

After all this recursion business we are going to finish with a very simple question? Why do we use recursion? The question is short and simple, but the answer is hardly easy. Let us start with *execution speed*. You may suspect that recursion executes faster than iteration. Speed is always desirable and that would certainly make recursion popular if greater speed is gained. Do you think that recursion is faster than iteration? Sorry ... good guess, but recursion is actually slower than iteration. The reason is that calling methods to execute a series of program statements takes longer because it makes a jump away from the current memory sequence. The time penalty for calling a method is extremely slight, but a large number of method calls can make a difference. A method like the **bubbleSort**, which you saw in the last chapter, has a very very large number of method calls. Remember the Fibonacci Sequence problem. The excessive number of recursive calls slowed the execution speed down tremendously.

Well if speed is not the reason for recursion then what might it be? How about *memory*? All these recursive methods seem to be quite short, and they are certainly shorter than the iterative methods. Yes, it must be memory. There is bad news again my good friends. Recursion uses more memory. The smaller code of the recursive methods may occupy less space than code for iterative methods, but there are other factors to consider. Our friendly stack stores local method information and controls the execution sequence of method calls. Lots of recursive calls can result in lots of unfinished business, which requires lots of memory to store until the program execution returns later to clean it up.

You are now officially stumped. Two of the best reasons for doing something different in a program, *speed* and *memory*have been eliminated from contention.

What other reasons are there for doing something in a program? Is it *readability* perhaps? You may remember that *readability* is a good thing for program debugging and future program enhancement.

Can it be that recursion is more *readable*? The answer to this question is a little tricky. There are plenty of people who will claim that recursion is natural and thus more readable. There is a good argument to be made, but there are also plenty of students, new to recursion, who will need some more exposure before all this recursive business is natural to them. And whether or not recursion is readable to you does not matter because it is not the reason for using recursion. You did have a preview of the reason for using recursion when you looked at the *Tower of Hanoi*  problem.

|  |
| --- |
| **The Reason for Using Recursion** |
| Recursion is preferred over iteration when it is easier to  write program code recursively. |

That is it? Yes, my good students that is precisely the reason. You may think that iteration is simpler than recursion, and it may very well be that way for methods like **skip**, **gcf**, **fact** and other small methods. Those method examples worked well to demonstrate recursive techniques. In reality there is little need to use a recursive **gcf** method or any of the others early method examples.

But think about the *Tower of Hanoi*. You were not shown how to do that problem iteratively and the recursion did seem pretty tricky. Yes, I agree, that problem was certainly tougher than the **fact** method. The point is that it is easier recursively than it is iteratively. Try it. See if you can create an iterative solution for the *Tower of Hanoi* problem.

**The Grid Problem Revisited**

At the beginning of this chapter you were challenged to solve a special problem in an iterative manner. You may or may not have tried to solve this problem, but the Grid problem is shown in this chapter as a problem that is much easier to do recursively than iteratively. The problem is repeated on the next page.

As you look as the problem specifications again, think how you would solve this problem iteratively. Many students might argue that they do not have a clue how to solve the problem iteratively and for that matter they are equally clueless how to solve the problem recursively. Basically, you consider this one nasty problem.

It is true that the recursive solution requires a considerable level of recursive sophistication. At the same time as you look at the problem specifications, and then the solution, I hope that you will be impressed that recursion has a solution that is pretty neat. Then continue to think how this might be done iteratively.

**The Grid Problem**

An image can be represented as a grid of black and white cells. Two cells in an image are part of the same object if each is black and there is a sequence of moves from one cell to the other, where each move is either horizontal or vertical to an adjacent black cell. For example, the diagram below represents an image that contains four separate objects, one of them consisting of a single cell.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

Complete method **alterGrid** using the header below, such that all cells in the same object as **grid[row][col]** are set to **white**; otherwise **grid** is unchanged. **grid** is a two-dimensional array of **boolean** elements. An element value of **true** means a **black** cell and an element value of **false** means a **white** cell.

**public void alterGrid(boolean grid[][], int row, int col)**

The solution to this problem is slightly altered from the original specifications. It is a little tricky to create black and white squares in a grid. However, the spirit of the problem can be simulated with text characters, using **X**s and **O**s.

I will leave you to ponder over the solution of the problem. It is recursive. It is certainly elegant. Is this elegance starting to make sense? You may need to do some tracing to see what is happening. But it is now your show. You have a program to work with, and you have the solution. This will be a good way to finish the chapter by carefully digesting an excellent example of a problem that was meant to be done recursively. The program may seem long and complex, but the actual recursive **alter** method, which solves the problem, is quite short and shown at the end of the program.

There are four statements that make recursive calls. Think why that is necessary and what it does to the program’s execution. Program **Java1914.java**, in figure 19.28, solves the Grid problem.

## Figure 19.28

|  |
| --- |
| // Java1914.java  // Recursive solution to the black/white "Grid" problem  import java.util.\*;  public class Java1914  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter row value ===>> ");  int row = input.nextInt();  System.out.print("Enter col value ===>> ");  int col = input.nextInt();  boolean grid[][] = new boolean[11][11];    create(grid);  display(grid);  alter(grid,row,col);  display(grid);  }  public static void create(boolean grid[][])  {  Random rand = new Random(12345);  for (int r = 1; r <= 10; r++)  for (int c = 1; c <= 10; c++)  {  int rndInt = rand.nextInt(2);  if (rndInt == 1)  grid[r][c] = true;  else  grid[r][c] = false;  }  }  public static void display(boolean grid[][])  {  System.out.println();  for (int r = 1; r <= 10; r++)  {  for (int c = 1; c <= 10; c++)  if (grid[r][c])  System.out.print("X ");  else  System.out.print("O ");  System.out.println();  }  }  **public static void alter(boolean grid[][], int r, int c)**  **{**  **if ((r >= 1) && (r <= 10) && (c >= 1) && (c <= 10))**  **if (grid[r][c])**  **{**  **grid[r][c] = false;**  **alter(grid,r-1,c);**  **alter(grid,r+1,c);**  **alter(grid,r,c-1);**  **alter(grid,r,c+1);**  **}**  **}**  } |

**Figure 19.28 Continued**

|  |
| --- |
| **Java1914.java Output #1**  Enter row value ===>> 1  Enter col value ===>> 1  O X X O X X O X O X  O O X O X O O X X O  X O O X X O X X O X  O O O O X O O O X X  O X O X X O O O X X  O O X X O X X X O X  O X X O O O O X X X  X X O O X O O X O O  O O O X X X X X O X  O X O O O O X O O X  O X X O X X O X O X  O O X O X O O X X O  X O O X X O X X O X  O O O O X O O O X X  O X O X X O O O X X  O O X X O X X X O X  O X X O O O O X X X  X X O O X O O X O O  O O O X X X X X O X  O X O O O O X O O X |

**Figure 19.28 Continued**

|  |
| --- |
| **Java1914.java Output #2**  Enter row value ===>> 2  Enter col value ===>> 2  X X X O X X O X X X  X X O O X X O X X O  X O O O O O O X X O  O O O O O O X O X X  X X X X X X X X O X  O X O X O X O O O X  X O O X O O X O O O  O O X O O O O X O X  O O O X X X O X O O  X X O O X X X O X O  O O O O X X O X X X  O O O O X X O X X O  O O O O O O O X X O  O O O O O O X O X X  X X X X X X X X O X  O X O X O X O O O X  X O O X O O X O O O  O O X O O O O X O X  O O O X X X O X O O  X X O O X X X O X O |

**Figure 19.28 Continued**

|  |
| --- |
| **Java1914.java Output #3**  Enter row value ===>> 5  Enter col value ===>> 5  X X X O X X X X X O  O X O X X O O X X X  X O X X X X O O O O  O O O O X O X O X O  O X O X X X X X X O  X O O X X O X O X O  X X O O O X O O X X  O X O X O X O X X O  X X X O X O X O X O  X O X X X O O O X O  X X X O O O O O O O  O X O O O O O O O O  X O O O O O O O O O  O O O O O O O O O O  O X O O O O O O O O  X O O O O O O O O O  X X O O O X O O O O  O X O X O X O O O O  X X X O X O X O O O  X O X X X O O O O O |

**19.11 Summary**

Recursion is the process whereby a method calls itself. This action simulates the actions of loop control structures without using **for**, **while** or **do while**.

All recursive methods require a special condition that stops the recursive calls. This condition is the *exit* of the method and it is normally called the *base case*. Without an exit the recursive process continuous until some runtime error stops the program execution.

Recursive calls can be placed in different method locations. If the recursive call occurs at the beginning of the method you have *pre-recursion* and if the recursive call occurs at the end of the method, you have *post-recursion* or *tail-recursion*. A method with pre-recursion will interrupt the completion of the method execution and place any unfinished business on the stack. The result is that execution is performed in a last in, first out (*LIFO*) sequence. This reverse sequence is especially noticed when each recursive call generates monitor output.

Iterative methods can also be written iteratively. Frequently, it is necessary to add additional parameters to recursive methods. This happens because any local variables, which are initialized in an iterative method will not work properly in a recursive method. Removing the local variable from the method body and placing it in the parameter heading solves this problem. Examples of methods that require additional parameters for recursive solutions are the Linear Search and Binary Search methods.

Recursive methods require less source code than iterative methods. The briefer code does not necessarily mean that recursive methods execute faster than their iterative cousins. This point is especially proven by the Fibonacci sequence. The solution of the Fibonacci sequence uses two recursive calls in a single program statement. This type of recursive solution doubles the required processing with each recursive call and slows down execution very dramatically.

It may seem that recursion is not much of an improvement over the previously established loop control structures. Recursion is slower, uses more memory and many students will argue that it is far less readable. The reality is that there are complex methods that are easier to solve recursively than iteratively. Two such methods were presented in this chapter. They are the Tower of Hanoi and the Grid puzzles.

This chapter represented an introduction to recursion. The second year course will revisit recursion and use recursive solution for very practical applications.