

Chapter 12

Recursion

12.1 Thinking Recursively

- A Nonprogramming Example
- An Iterative Solution Converted to Recursion
- Structure of Recursive Solutions

12.2 A Better Example of Recursion

- Mechanics of Recursion

12.3 Recursive Functions and Data

- Integer Exponentiation
- Greatest Common Divisor
- Directory Crawler
- Helper Methods

12.4 Recursive Graphics

12.5 Recursive Backtracking

- A Simple Example: Traveling North/East
- 8 Queens Puzzle
- Solving Sudoku Puzzles

12.6 Case Study: Prefix Evaluator

- Infix, Prefix, and Postfix Notation
- Evaluating Prefix Expressions
- Complete Program

Introduction

This chapter focuses on a programming technique known as *recursion* that allows us to solve certain complex problems in a highly elegant manner. The chapter begins by comparing recursion with the problem-solving techniques you already know. Then it discusses the low-level mechanics that make recursion work in Java. Finally, we examine a number of problems that are easily expressed using this technique.

Recursion turns out to have a surprising range of useful applications, including recursive graphics that are known as fractals. But programming recursively also requires some special techniques that we'll have to explore. Recursive programming also requires a different mind-set in general, so the chapter explores a large set of example problems to reinforce this new way of thinking.

12.1 Thinking Recursively

The problem-solving techniques we have employed so far fall under the heading of classical *iteration*, also known as the *iterative approach*.

Iteration (Iterative)

A programming technique in which you describe actions to be repeated using a loop.

In this chapter we will explore a new technique known as *recursion*.

Recursion (Recursive)

A programming technique in which you describe actions to be repeated using a method that calls itself.

You have spent so much time writing solutions iteratively that it will take a while to get used to thinking about problems recursively. This chapter will help you get acclimated.

A Nonprogramming Example

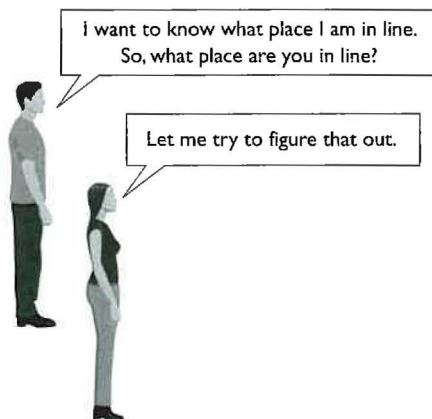
If you're standing in line, you might wonder what position you're in. Are you number 10 in the line? Number 20? How would you find out?

Most people would solve this problem iteratively, by counting the people in the line: one, two, three, and so on. This approach is like a `while` loop that continues while there are more people left to count. The iterative approach is a natural one, but it has some limitations. For example, what if the person in front of you is taller than you? Will you be able to see past that person to count all the people in front of him? And what if the line goes around the block and you can't see around the corner to count the people there?

Can you think of another way to determine your position in the line? To think about the problem recursively, you have to imagine that all the people standing in line work together to solve the problem: Instead of having one person do all of the counting, each person is responsible for a little piece.

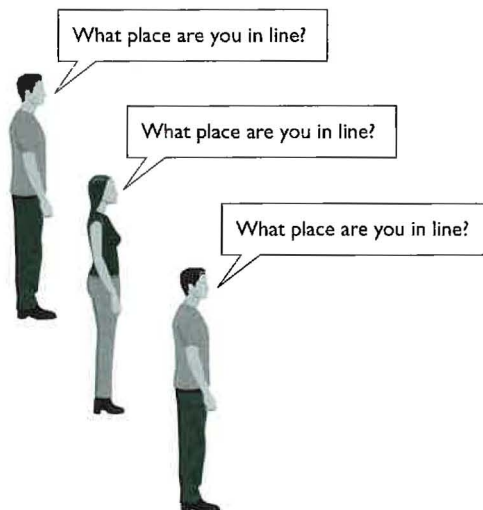
One cooperative approach would be to ask the person in front of you what your place in line is. That person might ask another person, who might ask another person. But that doesn't help much, because it just leads to a bunch of people saying, "This guy wants to know what place he is in line. Does anyone know?" Someone would probably eventually start counting and solve the problem iteratively.

You have to make the problem simpler. Instead of asking the person in front of you what place *you* are in line, ask that person what place *he* or *she* is in line:

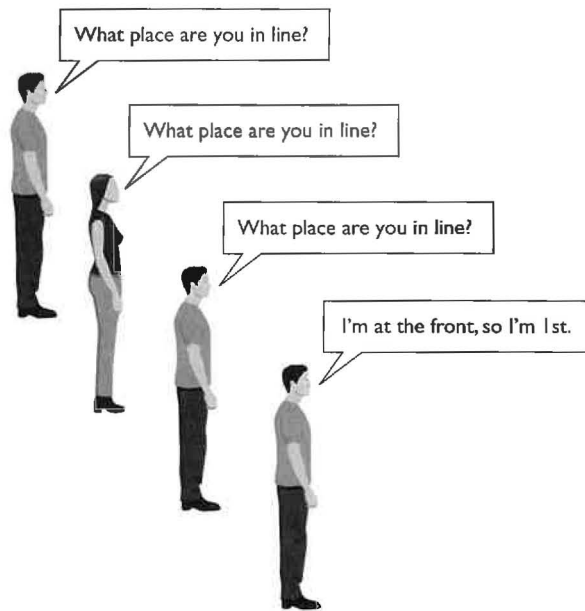


The key difference is that the person in front of you is closer to the front of the line. Suppose, for example, that you're 4th in line. The person in front of you is 3rd in line, which is closer to the front. But notice that you're asking the person in front of you to think about the exact same *kind* of question you're considering: You're both trying to figure out your places in line. That's where recursion comes in—the problem *recurs* because each of you wants to answer the same question.

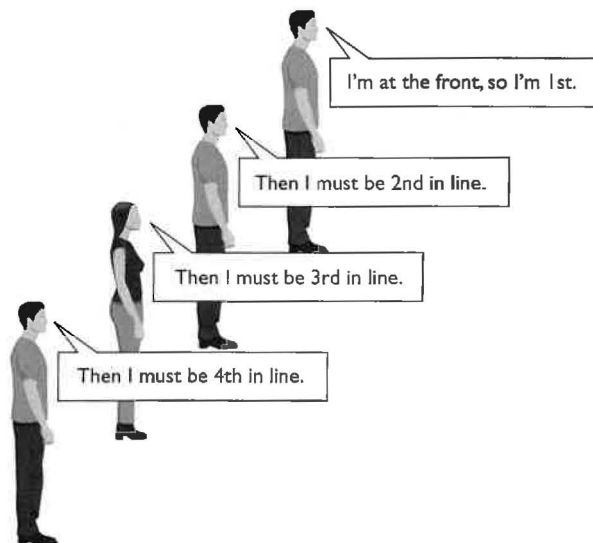
The idea is to set up a chain reaction of people, all asking the person in front the same question:



This process has to end eventually, when someone asks the person who is first in line:



At this point you've reached what is sometimes referred to as the *bottom* of the recursion. You've gotten a bunch of people involved in collectively solving the problem, and you've finally reached a point where you can start assembling the answer. The person at the front is in position 1. That means the person just before is at position 2, and the person just before that person is at position 3, and so on. Once you reach the bottom of the recursion, you *unwind* it to figure out the answer to your initial problem:



These diagrams included just 4 individuals for the sake of brevity, but this process would still work even if there were 30 or even 300 people in the line.

One of the key aspects to notice here is that recursion involves many cooperating entities, each of which solves a little bit of the problem. Instead of one person doing all of the counting, each individual asks one question as we go toward the front of the line and answers one question as we come back down the line.

In programming, the iterative solution of having one person do all the counting is like having a loop that repeats some action. The recursive solution of having many people each do a little bit of work translates into many different method calls, each of which performs a little bit of work. Let's look at an example of how a simple iterative solution can be turned into a recursive solution.

An Iterative Solution Converted to Recursion

As a first example, we will explore a problem that has a simple iterative solution. It won't be a very impressive use of recursion because the problem is easily solved with iteration. But exploring a problem that has a straightforward iterative solution allows us to compare the two solutions.

Suppose you want to create a method called `writeStars` that will take an integer parameter `n` and will produce a line of output with exactly `n` stars on it. You can solve this problem with a simple `for` loop:

```
public static void writeStars(int n) {
    for (int i = 1; i <= n; i++) {
        System.out.print("*");
    }
    System.out.println();
}
```

The action being repeated here is the call on `System.out.print` that prints a star. To write the stars recursively, you need to think about different cases. You might ask the method to produce a line with 10 stars, 20 stars, or 50 stars. Of all of the possible star-writing tasks you might ask it to perform, which is the simplest?

Students often answer that printing a line of one star is very easy and they're right that it's easy. But there is a task that is even easier. Printing a line of zero stars requires almost no work at all. You can create such a line by calling `System.out.println`, so you can begin your recursive definition with a test for this case:

```
public static void writeStars(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        ...
    }
}
```

The code in the `else` part will deal with lines that have more than zero stars on them. Your instinct will probably be to fill in the `else` part with the `for` loop shown earlier, but you'll have to fight the instinct to solve the entire problem that way. To solve this second part of the problem, it is important to think about how you can do just a small amount of work that will get you closer to the solution. If the number of stars is greater than zero, you know you have to print at least one star, so you can add that action to the code:

```
public static void writeStars(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        System.out.print("*");
        // what is left to do?
        ...
    }
}
```

At this point in the process you have to make a leap of faith: You have to believe that recursion actually works. Once you've printed a single star, what's left to do? The answer is that you want to write $(n - 1)$ more stars, along with a `println`. In other words, after writing one star, the task that remains is to write a line of $(n - 1)$ stars. You may think, "If only I had a method that would produce a line of $(n - 1)$ stars, I could call that method." But you *do* have such a method—the method you're writing. So, after your method writes a single star, you can call the `writeStars` method itself to complete the line of output:

```
public static void writeStars(int n) {
    if (n == 0) {
        System.out.println();
    } else {
        System.out.print("*");
        writeStars(n - 1);
    }
}
```

Many novices complain that this seems like cheating. You're supposed to be writing the method called `writeStars`, so how can you call `writeStars` from inside `writeStars`? Welcome to the world of recursion.

In the earlier example, we talked about people standing in a line and solving a problem together. To understand a recursive method like `writeStars`, it is useful to imagine that each method invocation is like a person in the line. The key insight is that there isn't just one person who can do the `writeStars` task; there's an entire army of people, each of whom can do the task.

Let's think about what happens when you call the method and request a line of three stars:

```
writeStars(3);
```

Imagine that you're calling up the first person from the `writeStars` army and saying, "I want a line of three stars." That person looks at the code in the method and sees that the way to write a line of three stars is to execute the following lines:

```
System.out.print("*");
writeStars(2);
```

In other words, the first member of the army writes a star and calls up the next member of the army to write a line of two stars, and so on down the line. Just as in the earlier example you had a series of people figuring out their places in line, now you have a series of people each printing one star and then calling on someone else to write the rest of the line. In the line example, you eventually reached the person at the front of the line. In this case, you eventually reach a request to write a line of zero stars, which leads you into the `if` branch rather than the `else` branch. At this point, you complete the task with a simple `println`.

Here is a trace of the calls that would be made to print the line:

```
writeStars(3); // n > 0, execute else
    System.out.print("*");
    writeStars(2); // n > 0, execute else
        System.out.print("*");
        writeStars(1); // n > 0, execute else
            System.out.print("*");
            writeStars(0); // n == 0, execute if
                System.out.println();
```

A total of four different calls are made on the method. Continuing the analogy, you could say that four members of the army are called up to solve the task together. Each one solves a star-writing task, but the tasks are slightly different (three stars, two stars, one star, zero stars). This is similar to the example in which the various people standing in line were all answering the same question but were solving slightly different problems because their positions in line were different.

Structure of Recursive Solutions

Writing recursive solutions requires you to make a certain leap of faith, but there is nothing magical about recursion. Let's look a bit more closely at the structure of a recursive solution. The following method is not a solution to the task of writing a line of `n` stars:

```
// does not work
❏ public static void writeStars(int n) {
    writeStars(n);
}
```

This version never finishes executing, a phenomenon called *infinite recursion*. For example, if you ask the method to write a line of 10 stars, it tries to accomplish that by asking the method to write a line of 10 stars, which asks the method to write a line of 10 stars, which asks the method to write a line of 10 stars, and so on. This solution is the recursive equivalent of an infinite loop.

Every recursive solution that you write will have two key ingredients: a *base case* and a *recursive case*.

Base Case

A case within a recursive solution that is so simple that it can be solved directly without a recursive call.

Recursive Case

A case within a recursive solution that involves reducing the overall problem to a simpler problem of the same kind that can be solved by a recursive call.

Here is the `writeStars` method again, with comments indicating the base case and recursive case:

```
public static void writeStars(int n) {  
    if (n == 0) {  
        // base case  
        System.out.println();  
    } else {  
        // recursive case  
        System.out.print("*");  
        writeStars(n - 1);  
    }  
}
```

The base case is the task of writing a line of zero stars. This task is so simple that it can be done immediately. The recursive case is the task of writing lines with one or more stars. To solve the recursive case, you begin by writing a single star, which reduces the remaining task to that of writing a line of $(n - 1)$ stars. This is the task that the `writeStars` method is designed to solve and it is simpler than the original task, so you can solve it by making a recursive call.

As an analogy, suppose you're at the top of a ladder with n rungs on it. If you have a way to get from one rung to the one below and if you can recognize when you've reached the ground, you can handle a ladder of any height. Stepping from one rung to the one below is like the recursive case in which you perform some small amount of work that reduces the problem to a simpler one of the same form (get down from rung $(n - 1)$ versus get down from rung n). Recognizing when you reach the ground is like the base case that can be solved directly (stepping off the ladder).

Some problems involve multiple base cases and some problems involve multiple recursive cases, but there will always be at least one of each case in a correct recursive solution. If you are missing either, you run into trouble. Without the ability to step down from one rung to the one below, you'd be stuck at the top of the ladder. Without the ability to recognize when you reach the ground, you'd keep trying to step down onto another rung even when there were no rungs left in the ladder.

Keep in mind that your code can have infinite recursion even if it has a proper recursive case. Consider, for example, this version of `writeStars`:

```
// does not work
public static void writeStars(int n) {
    System.out.print("*");
    writeStars(n - 1);
}
```

This version correctly reduces from the case of n to the case of $n - 1$, but it has no base case. As a result, it goes on infinitely. Instead of stopping when it reaches the task of writing zero stars, it instead recursively tries to write -1 stars, then -2 stars, then -3 stars, and so on.

Because recursive solutions include some combination of base cases and recursive cases, you will find that they are often written with `if/else` statements, nested `if` statements, or some minor variation thereof. You will also find that recursive programming generally involves a case analysis, in which you categorize the possible forms the problem might take into different cases and write a solution for each case.

12.2 A Better Example of Recursion

Solving the `writeStars` task with recursion may have been an interesting exercise, but it isn't a very compelling example. Let's look in detail at a problem in which recursion simplifies the work to be done.

Suppose you have a scanner that is tied to an external input file and you want to print the lines of the file in reverse order. For example, the file might contain the following four lines of text:

```
this
is
fun
no?
```

Printing these lines in reverse order would produce this output:

```
no?
fun
is
this
```


To perform this task iteratively, you'd need some kind of data structure for storing the lines of text, such as an `ArrayList<String>`. However, recursion allows you to solve the problem without using a data structure.

Remember that recursive programming involves thinking about cases. What would be the simplest file to reverse? A one-line file would be fairly easy to reverse, but it would be even easier to reverse an empty file. So, you can begin writing your method as follows:

```
public static void reverse(Scanner input) {
    if (!input.hasNextLine()) {
        // base case (empty file)
        ...
    } else {
        // recursive case (nonempty file)
        ...
    }
}
```

In this problem, the base case is so simple that there isn't anything to do. An empty file has no lines to reverse. Thus, in this case it makes more sense to turn around the `if/else` statement so that you test for the recursive case. That way you can write a simple `if` statement that has an implied "else there is nothing to do":

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        ...
    }
}
```

Again, the challenge is to solve only a little bit of the problem. How do you take just one step that will get you closer to completing the task? You can read one line of text from the file:

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        ...
    }
}
```

For the sample file, this code would read the line "this" into the variable `line` and leave you with the following three lines of text in the `Scanner`:

```
is
fun
no?
```

Recall that your aim is to produce the following overall output:

```
no?
fun
is
this
```

You might be asking yourself questions like, "Is there another line of input to process?" But that's not recursive thinking. If you're thinking recursively, you'll be thinking about what a call on the method will get you. Since the `Scanner` is positioned in front of the three lines "is", "fun", and "no?", a call on `reverse` should read in those lines and produce the first three lines of output that you're looking for. If that works, you'll only have to write out the line "this" afterward to complete the output.

This is where the leap of faith comes in—you have to believe that the `reverse` method actually works. If it does, this code can be completed as follows:

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        reverse(input);
        System.out.println(line);
    }
}
```

This code does work. To reverse a sequence of lines, simply read in the first one, reverse the others, and then write out the first one.

Mechanics of Recursion



Novices seem to understand recursion better when they know more about the underlying mechanics that make it work. Before we examine a recursive method in detail, let's review how nonrecursive methods work. Consider the following simple program:

```
1 // Simple program that draws three triangles.
2 public class DrawTriangles {
3     public static void main(String[] args) {
4         drawTriangle();
5         drawTwoTriangles();
```

```
6     }
7
8     public static void drawTriangle() {
9         System.out.println("  *");
10        System.out.println(" ***");
11        System.out.println("*****");
12        System.out.println();
13    }
14
15    public static void drawTwoTriangles() {
16        drawTriangle();
17        drawTriangle();
18    }
19 }
```

The program prints three triangles:

```
*
***
*****

*
***
*****

*
***
*****
```

How do we describe the method calls that take place in this program? Imagine that each method call has been written on a different piece of paper. We begin program execution with the main method, so imagine that we're grabbing the sheet of paper with main on it:

```
public static void main(String[] args) {
    drawTriangle();
    drawTwoTriangles();
}
```

We then execute each of the statements in main in turn, from first to last. First we execute the call on drawTriangle:

```
public static void main(String[] args) {
→   drawTriangle();
    drawTwoTriangles();
}
```

We know that at this point, the computer stops executing main and turns its attention to the drawTriangle method. This step is analogous to picking up the piece of paper with drawTriangle written on it and placing it over the piece of paper with main on it:

```
public static void main(String[] args) {
    public static void drawTriangle() {
        System.out.println("  *");
        System.out.println(" ***");
        System.out.println("*****");
        System.out.println();
    }
}
```

Now we execute each of the statements in drawTriangle from first to last, then go back to main, removing the drawTriangle sheet. When we go back to main we will have finished the call on drawTriangle, so the next step is the call on drawTwoTriangles:

```
public static void main(String[] args) {
    drawTriangle();
    → drawTwoTriangles();
}
```

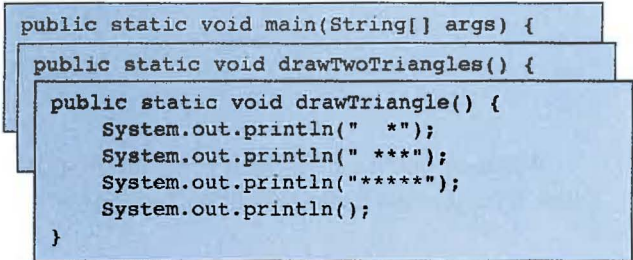
So we grab the piece of paper with drawTwoTriangles on it and place it over the paper with main on it:

```
public static void main(String[] args) {
    public static void drawTwoTriangles() {
        drawTriangle();
        drawTriangle();
    }
}
```

The first thing to do here is the first call on drawTriangle:

```
public static void main(String[] args) {
    public static void drawTwoTriangles() {
        → drawTriangle();
        drawTriangle();
    }
}
```


To execute this method, we take out the sheet of paper with `drawTriangle` on it and place it on top of the `drawTwoTriangles` sheet:



```
public static void main(String[] args) {  
    public static void drawTwoTriangles() {  
        public static void drawTriangle() {  
            System.out.println(" *");  
            System.out.println(" ***");  
            System.out.println("*****");  
            System.out.println();  
        }  
    }  
}
```

This diagram makes it clear that we started with the method `main`, which called the method `drawTwoTriangles`, which called the method `drawTriangle`. So, at this moment in time, three different methods are active. The one on top is the one that we are actively executing. Once we complete it, we'll go back to the one underneath, and once we finish that one, we'll go back to `main`. We could continue with this example, but you probably get the idea by now.

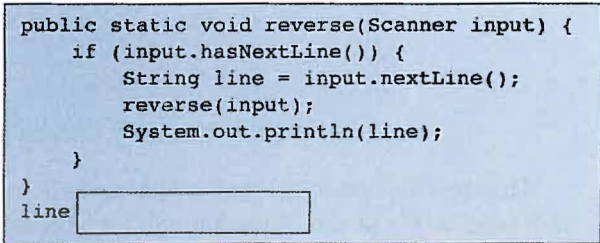
The idea of representing each method call as a piece of paper and putting each one on top of the others as it is called is a metaphor for Java's *call stack*.

Call Stack

The internal structure that keeps track of the sequence of methods that have been called.

If you envision the call stack as a stack of papers with the most recently called method on top, you'll have a pretty good idea of how it works.

Let's use the idea of the call stack to understand how the recursive file-reversing method works. To visualize the call stack, we need to put the method definition on a piece of paper:



```
public static void reverse(Scanner input) {  
    if (input.hasNextLine()) {  
        String line = input.nextLine();  
        reverse(input);  
        System.out.println(line);  
    }  
}  
line
```

Notice that the paper includes a place to store the value of the local variable `line`. This is an important detail.

Suppose that we call this method with the earlier sample input file, which contains the following four lines of text:

```
this  
is  
fun  
no?
```

When we call the method, it reads the first line of text into its `line` variable, and then it reaches the recursive call on `reverse`:

```
public static void reverse(Scanner input) {  
    if (input.hasNextLine()) {  
        String line = input.nextLine();  
        → reverse(input);  
        System.out.println(line);  
    }  
}  
line "this"
```

Then what happens? In the `DrawTriangles` program, we took the sheet of paper for the method being called and placed it on top of the current sheet of paper. But here we have the method `reverse` calling the method `reverse`. To understand what happens, you have to realize that each method invocation is independent of the others. We don't have only a single sheet of paper with the `reverse` method written on it; we have as many copies as we want. So, we can grab a second copy of the method definition and place it on top of the current one:

```
public static void reverse(Scanner input) {  
    public static void reverse(Scanner input) {  
        if (input.hasNextLine()) {  
            String line = input.nextLine();  
            reverse(input);  
            System.out.println(line);  
        }  
    }  
    line 
```

This new version of the method has a variable of its own, called `line`, in which it can store a line of text. Even though the previous version (the one underneath this one) is in the middle of its execution, this new one is at the beginning of its execution. Think back to the analogy of being able to employ an entire army of people to write out a line of stars. Just as you could call on as many people as you needed to solve that problem, you can bring up as many copies of the `reverse` method as you need to solve this problem.

The second call on `reverse` reads another line of text (the second line, "is"). After the program reads the second line, it makes another recursive call on `reverse`:

```
public static void reverse(Scanner input) {  
    public static void reverse(Scanner input) {  
        if (input.hasNextLine()) {  
            String line = input.nextLine();  
            → reverse(input);  
            System.out.println(line);  
        }  
    }  
    line "is"
```

So Java sets aside this version of the method as well and brings up a third version:

```
public static void reverse(Scanner input) {  
    public static void reverse(Scanner input) {  
        public static void reverse(Scanner input) {  
            if (input.hasNextLine()) {  
                String line = input.nextLine();  
                reverse(input);  
                System.out.println(line);  
            }  
        }  
        line
```

Again, notice that this version has its own variable called `line` that is independent of the other variables called `line`. This version of the method also reads in a line (the third line, "fun") and reaches a recursive call on `reverse`:

```
public static void reverse(Scanner input) {  
    public static void reverse(Scanner input) {  
        public static void reverse(Scanner input) {  
            if (input.hasNextLine()) {  
                String line = input.nextLine();  
            → reverse(input);  
            System.out.println(line);  
        }  
    }  
    line "fun"
```


This brings up a fourth version of the method:

```
public static void reverse(Scanner input) {
    public static void reverse(Scanner input) {
        public static void reverse(Scanner input) {
            public static void reverse(Scanner input) {
                if (input.hasNextLine()) {
                    String line = input.nextLine();
                    reverse(input);
                    System.out.println(line);
                }
            }
        }
    }
    line
```

This version finds a fourth line of input ("no?"), so it reads that in and reaches the recursive call:

```
public static void reverse(Scanner input) {
    public static void reverse(Scanner input) {
        public static void reverse(Scanner input) {
            public static void reverse(Scanner input) {
                if (input.hasNextLine()) {
                    String line = input.nextLine();
                    → reverse(input);
                    System.out.println(line);
                }
            }
        }
    }
    line "no?"
```

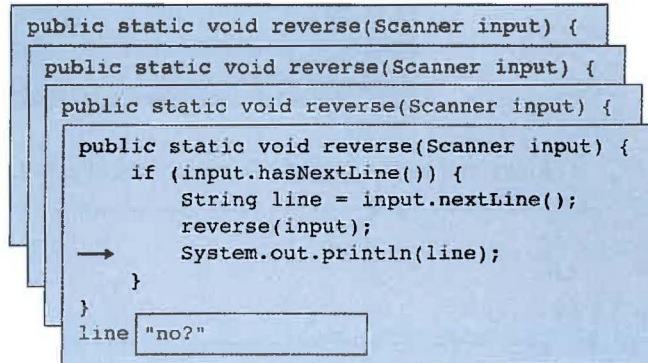
This call brings up a fifth version of the method:

```
public static void reverse(Scanner input) {
    public static void reverse(Scanner input) {
        public static void reverse(Scanner input) {
            public static void reverse(Scanner input) {
                public static void reverse(Scanner input) {
                    if (input.hasNextLine()) {
                        String line = input.nextLine();
                        reverse(input);
                        System.out.println(line);
                    }
                }
            }
        }
    }
    line
```

This version turns out to have the easy task, like the final person who was asked to print a line of zero stars. This time around the Scanner is empty

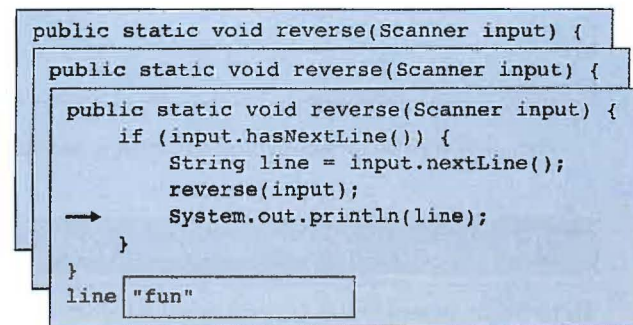
(`input.hasNextLine()` returns `false`). The program has reached the very important base case that stops this process from going on indefinitely. This version of the method recognizes that there are no lines to reverse, so it simply terminates.

Then what? Having completed this call, we throw it away and return to where we were just before executing the call:



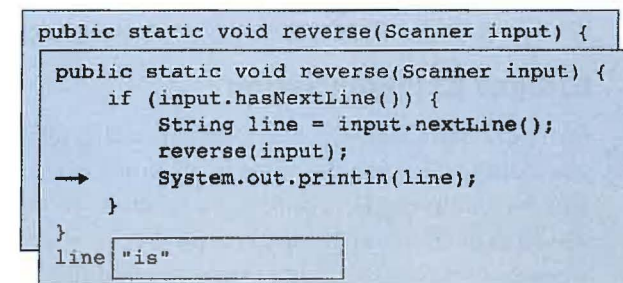
```
public static void reverse(Scanner input) {
    public static void reverse(Scanner input) {
        public static void reverse(Scanner input) {
            public static void reverse(Scanner input) {
                if (input.hasNextLine()) {
                    String line = input.nextLine();
                    reverse(input);
                    System.out.println(line);
                }
            }
        }
    }
    line "no?"
}
```

We've finished the call on `reverse` and are positioned at the `println` right after it, so we print the text in the `line` variable ("no?") and terminate. Where does that leave us? This method has been executed and we return to where we were just before:



```
public static void reverse(Scanner input) {
    public static void reverse(Scanner input) {
        public static void reverse(Scanner input) {
            if (input.hasNextLine()) {
                String line = input.nextLine();
                reverse(input);
                System.out.println(line);
            }
        }
    }
    line "fun"
}
```

We then print the current line of text, which is "fun", and this version also goes away:



```
public static void reverse(Scanner input) {
    public static void reverse(Scanner input) {
        if (input.hasNextLine()) {
            String line = input.nextLine();
            reverse(input);
            System.out.println(line);
        }
    }
    line "is"
}
```

Now we execute this `println`, for the text "is", and eliminate one more call:

```
public static void reverse(Scanner input) {  
    if (input.hasNextLine()) {  
        String line = input.nextLine();  
        reverse(input);  
        → System.out.println(line);  
    }  
}  
line "this"
```

Notice that we've written out three lines of text so far:

```
no?  
fun  
is
```

Our leap of faith was justified. The recursive call on `reverse` read in the three lines of text that followed the first line of input and printed them in reverse order. We complete the task by printing the first line of text, which leads to this overall output:

```
no?  
fun  
is  
this
```

Then this version of the method terminates, and the program has finished executing.

12.3 Recursive Functions and Data



Both of the examples of recursion we have studied so far have been action-oriented methods with a return type of `void`. In this section we will examine some of the issues that arise when you want to write methods that compute values and return a result. Such methods are similar to mathematical functions in that they accept a set of input values and produce a set of possible results. We'll also explore an example that involves manipulating recursive data and an example that requires a helper method.

Integer Exponentiation

Java provides a method called `Math.pow` that allows you to compute an exponent. If you want to compute the value of x^y , you can call `Math.pow(x, y)`. Let's consider how we could implement the `pow` method. To keep things simple, we'll limit ourselves to the domain of integers. But because we are limiting ourselves to integers, we have to recognize an important precondition of our method: We won't be able to compute negative exponents because the results would not be integers.

The method we want to write will look like the following:

```
// pre : y >= 0
// post: returns x^y
public static int pow(int x, int y) {
    ...
}
```

We could obviously solve this problem by writing a loop, but we want to explore how to write the method recursively. Again, we should start by thinking about different cases. What would be the easiest exponent to compute? It's pretty easy to compute x^1 , so that's a good candidate, but there is an even more basic case. The simplest possible exponent is 0. By definition, any integer to the 0 power is considered to be 1. So we can begin our solution with the following code:

```
public static int pow(int x, int y) {
    if (y == 0) {
        // base case with y == 0
        return 1;
    } else {
        // recursive case with y > 0
        ...
    }
}
```

In the recursive case we know that y is greater than 0. In other words, there will be at least one factor of x in the result. We know from mathematics that

$$x^y = x \cdot x^{y-1}$$

This equation expresses x to the y power in terms of x to a smaller power, $(y - 1)$. Therefore, it can serve as our recursive case. All we have to do is to translate it into its Java equivalent:

```
public static int pow(int x, int y) {
    if (y == 0) {
        // base case with y == 0
        return 1;
    } else {
        // recursive case with y > 0
        return x * pow(x, y - 1);
    }
}
```

This is a complete recursive solution. Tracing the execution of a recursive function is a little more difficult than using a void method, because we have to keep track of the values that are returned by each recursive call. The following is a trace of execution showing how we would compute 3^5 :

```

pow(3, 5) = 3 * pow(3, 4)
    pow(3, 4) = 3 * pow(3, 3)
        pow(3, 3) = 3 * pow(3, 2)
            pow(3, 2) = 3 * pow(3, 1)
                pow(3, 1) = 3 * pow(3, 0)
                    pow(3, 0) = 1
                pow(3, 1) = 3 * 1 = 3
            pow(3, 2) = 3 * 3 = 9
        pow(3, 3) = 3 * 9 = 27
    pow(3, 4) = 3 * 27 = 81
pow(3, 5) = 3 * 81 = 243

```

Notice that we make a series of six recursive calls in a row until we reach the base case of computing 3 to the 0 power. That call returns the value 1 and then the recursion unwinds, computing the various answers as it returns from each method call.

It is useful to think about what will happen if someone violates the precondition by asking for a negative exponent. For example, what if someone asked for `pow(3, -1)`? The method would recursively ask for `pow(3, -2)`, which would ask for `pow(3, -3)`, which would ask for `pow(3, -4)`, and so on. In other words, it would lead to an infinite recursion. In a sense, it's okay for this to occur, because the person calling the method should pay attention to the precondition and should not enter a negative exponent. But it's not much work for us to handle this case in a more elegant manner. Our solution is structured as a series of cases, so we can simply add a new case for illegal exponents:

```

public static int pow(int x, int y) {
    if (y < 0) {
        throw new IllegalArgumentException("negative exponent: " + y);
    } else if (y == 0) {
        // base case with y == 0
        return 1;
    } else {
        // recursive case with y > 0
        return x * pow(x, y - 1);
    }
}

```

One of the advantages of writing functions recursively is that if we can identify other cases, we can potentially make the function more efficient. For example, suppose that you want to compute 2^{16} . In its current form, the method will multiply 2 by 2 by 2 a total of 16 times. But we can do better than that. If y is an even exponent, then

$$x^y = (x^2)^{\frac{y}{2}}$$

So instead of computing 2^{16} , we can compute 4^8 , which is simpler. Adding this case to our method is relatively easy:


```

public static int pow(int x, int y) {
    if (y < 0) {
        throw new IllegalArgumentException("negative exponent: " + y);
    } else if (y == 0) {
        // base case with y == 0
        return 1;
    } else if (y % 2 == 0) {
        // recursive case with y > 0, y even
        return pow(x * x, y / 2);
    } else {
        // recursive case with y > 0, y odd
        return x * pow(x, y - 1);
    }
}

```

This version of the method is more efficient than the original. The following is a trace of execution for computing 2^{16} :

```

pow(2, 16) = pow(4, 8)
    |
    | pow(4, 8) = pow(16, 4)
    |   |
    |   | pow(16, 4) = pow(256, 2)
    |   |   |
    |   |   | pow(256, 2) = pow(65536, 1)
    |   |   |   |
    |   |   |   | pow(65536, 1) = 65536 * pow(65536, 0)
    |   |   |   |   |
    |   |   |   |   | pow(65536, 0) = 1
    |   |   |   |   |   |
    |   |   |   |   |   | pow(65536, 1) = 65536 * 1 = 65536
    |   |   |   |   |   |   |
    |   |   |   |   |   |   | pow(256, 2) = 65536
    |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   | pow(16, 4) = 65536
    |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   | pow(4, 8) = 65536
    |   |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |   | pow(2, 16) = 65536

```

Without the special case for even exponents, this call would have required 17 different calls on pow (16 recursive cases and one base case).

Greatest Common Divisor

In mathematics, we often want to know the largest integer that goes evenly into two different integers, which is known as the *greatest common divisor* (or GCD) of the two integers. Let's explore how to write a GCD method recursively.

For now, let's not worry about negative values of x and y . We want to write the following method:

```

// pre: x >= 0, y >= 0
// post: returns the greatest common divisor of x and y
public static int gcd(int x, int y) {
    ...
}

```

To introduce some variety, let's try to figure out the recursive case first and then figure out the base case. Suppose, for example, that we are asked to compute the GCD of 20 and 132. The GCD is 4, because 4 is the largest integer that goes evenly into both numbers.

There are many ways to compute the GCD of two numbers. One of the most efficient algorithms dates back at least to the time of Euclid and perhaps even farther. This algorithm eliminates any multiples of the smaller integer from the larger integer. In the case of 20 and 132, we know that

$$132 = 20 \cdot 6 + 12$$

There are six multiples of 20 in 132, with a remainder of 12. Euclid's algorithm says that we can ignore the six multiples of 20 and just focus on the value 12. In other words, we can replace 132 with 12:

$$\text{gcd}(132, 20) = \text{gcd}(12, 20)$$

We haven't figured out the base case yet, but no matter what the base case ends up being, we're making progress if we can reduce the numbers using Euclid's algorithm. When you're dealing with nonnegative integers, you can't reduce them forever.

The proof of this principle is beyond the scope of this book, but that is the basic idea. This algorithm is easy to express in Java terms because the mod operator gives us the remainder when one number is divided by another. Expressing this principle in general terms, we know that

$$\text{gcd}(x, y) = \text{gcd}(x \% y, y) \quad \text{when } y > 0$$

Again, the proof is beyond the scope of this book, but given this basic principle we can produce a recursive solution to the problem. We might try to write the method as follows:

```
public static int gcd(int x, int y) {
    if (...) {
        // base case
        ...
    } else {
        // recursive case
        return gcd(x % y, y);
    }
}
```

This isn't a bad first attempt, but it has a problem: It's not enough for the solution to be mathematically correct; we also need our recursive solution to keep reducing the overall problem to a simpler problem. If we start with the numbers 132 and 20, the method makes progress on the first call, but then it starts repeating itself:

$$\begin{aligned} \text{gcd}(132, 20) &= \text{gcd}(12, 20) \\ \text{gcd}(12, 20) &= \text{gcd}(12, 20) \end{aligned}$$

```
gcd(12, 20) = gcd(12, 20)
gcd(12, 20) = gcd(12, 20)
...
```

This pattern will lead to infinite recursion. The Euclidean trick helped the first time around, because for the first call x was greater than y (132 is greater than 20). But the algorithm makes progress only if the first number is larger than the second number.

Here is the line of code that is causing the problem:

```
return gcd(x % y, y);
```

When we compute $(x \% y)$, we are guaranteed to get a result that is smaller than y . That means that on the recursive call, the first value will always be smaller than the second value. To make the algorithm work, we need the opposite to be true. We can achieve this goal simply by reversing the order of the arguments:

```
return gcd(y, x % y);
```

On this call we are guaranteed to have a first value that is larger than the second value. If we trace this version of the method for computing the GCD of 132 and 20, we get the following sequence of calls:

```
gcd(132, 20) = gcd(20, 12)
gcd(20, 12) = gcd(12, 8)
gcd(12, 8) = gcd(8, 4)
gcd(8, 4) = gcd(4, 0)
...
```

At this point we have to decide what the GCD of 4 and 0 is. It may seem strange, but the answer is 4. In general, $\text{gcd}(n, 0)$ is n . Obviously, the GCD can't be any larger than n , and n goes evenly into n . But n also goes evenly into 0, because 0 can be written as an even multiple of n : $(0 * n) = 0$.

This observation leads us to the base case. If y is 0, the GCD is x :

```
public static int gcd(int x, int y) {
    if (y == 0) {
        // base case with y == 0
        return x;
    } else {
        // recursive case with y > 0
        return gcd(y, x % y);
    }
}
```

This base case also solves the potential problem that the Euclidean formula depends on y not being 0. However, we still have to think about the case in which

either or both of x and y are negative. We could keep the precondition and throw an exception when this occurs, but it is more common in mathematics to return the GCD of the absolute value of the two values. We can accomplish this by including one extra case for negatives:

```
public static int gcd(int x, int y) {
    if (x < 0 || y < 0) {
        // recursive case with negative value(s)
        return gcd(Math.abs(x), Math.abs(y));
    } else if (y == 0) {
        // base case with y == 0
        return x;
    } else {
        // recursive case with y > 0
        return gcd(y, x % y);
    }
}
```

Common Programming Error

Infinite Recursion

Everyone who uses recursion to write programs eventually accidentally writes a solution that leads to infinite recursion. For example, the following is a slight variation of the `gcd` method that doesn't work:

```
// flawed definition
public static int gcd(int x, int y) {
    if (x <= 0 || y <= 0) {
        // recursive case with negative value(s)
        return gcd(Math.abs(x), Math.abs(y));
    } else if (y == 0) {
        // base case with y == 0
        return x;
    } else {
        // recursive case with y > 0
        return gcd(y, x % y);
    }
}
```

This solution is just slightly different than the one we wrote previously. In the test for negative values, this code tests whether x and y are less than or equal to 0.

Continued on next page

Continued from previous page

The original code tests whether they are strictly less than 0. It doesn't seem like this variation should make much difference, but it does. If we execute this version of the code to solve our original problem of finding the GCD of 132 and 20, the program produces many lines of output that look like the following:

```
at Bug.gcd(Bug.java:9)
at Bug.gcd(Bug.java:9)
at Bug.gcd(Bug.java:9)
at Bug.gcd(Bug.java:9)
at Bug.gcd(Bug.java:9)
at Bug.gcd(Bug.java:9)
at Bug.gcd(Bug.java:9)
```

The first time you see this, you are likely to think that something has broken on your computer because you will get so many lines of output. The number of lines of output will vary from one system to another, but it's likely to be hundreds, if not thousands. If you scroll all the way back up, you'll see that the output begins with this message:

```
Exception in thread "main" java.lang.StackOverflowError
    at Bug.gcd(Bug.java:9)
    at Bug.gcd(Bug.java:9)
    at Bug.gcd(Bug.java:9)
    ...
```

Java is letting you know that the call stack has gotten too big. Why did this happen? Remember the trace of execution for this case:

```
gcd(132, 20) = gcd(20, 12)
    gcd(20, 12) = gcd(12, 8)
        gcd(12, 8) = gcd(8, 4)
            gcd(8, 4) = gcd(4, 0)
                ...
```

Consider what happens at this point, when we call `gcd(4, 0)`. The value of `y` is 0, which is our base case, so normally we would expect the method to return the value 4 and terminate. But the method begins by checking whether either `x` or `y` is less than or equal to 0. Since `y` is 0, this test evaluates to `true`, so the method makes a recursive call with the absolute values of `x` and `y`. But the absolute values of 4 and 0 are 4 and 0. So the method decides that `gcd(4, 0)` must be equal to `gcd(4, 0)`, which must be equal to `gcd(4, 0)`:

Continued on next page

Continued from previous page

```
gcd(132, 20) = gcd(20, 12)
    gcd(20, 12) = gcd(12, 8)
        gcd(12, 8) = gcd(8, 4)
            gcd(8, 4) = gcd(4, 0)
                gcd(4, 0) = gcd(4, 0)
                    gcd(4, 0) = gcd(4, 0)
                        gcd(4, 0) = gcd(4, 0)
                            gcd(4, 0) = gcd(4, 0)
                                ...
```

In other words, this version generates infinitely many recursive calls. Java allows you to make a lot of recursive calls, but eventually it runs out of space. When it does, it gives you a back trace to let you know how you got to the error. In this case, the back trace is not nearly as helpful as usual because almost all of the calls will involve the infinite recursion.

Again, think in terms of stacking pieces of paper on top of each other as methods are called. You'd wind up with a stack containing hundreds or even thousands of sheets, and you would have to look back through all of these to find the problem.

To handle these situations, you have to look closely at the line number to see which line of your program generated the infinite recursion. In this simple case, we know that it is the recursive call for negative *x* and *y* values. That alone might be enough to allow us to pinpoint the error. If the problem isn't obvious, though, you might need to include `println` statements to figure out what is going on. For example, in this code, we could add a `println` just before the recursive call:

```
public static int gcd(int x, int y) {
    if (x <= 0 || y <= 0) {
        // recursive case with negative value(s)
        System.out.println("x = " + x + " and y = " + y);
        return gcd(Math.abs(x), Math.abs(y));
    } else if (y == 0) {
        ...
    }
}
```

When we run the program with that `println` in place, the code produces hundreds of lines of output of the form:

```
x = 4 and y = 0
```

If we examine that case closely, we'll see that we don't have negative values and will realize that we have to fix the test we are using.

Directory Crawler

Recursion is particularly useful when you're working with data that is itself recursive. For example, consider how files are stored on a computer. Each file is kept in a folder or directory. But directories can contain more than just files: Directories can contain other directories, those inner directories can contain directories, and even those directories can contain directories. Directories can be nested to an arbitrary depth. This storage system is an example of recursive data.

In Chapter 6, you learned to use `File` objects to keep track of files stored on your computer. For example, if you have a file called `data.txt`, you can construct a `File` object that can be used to get information about that file:

```
File f = new File("data.txt");
```

Several useful methods for the `File` class were introduced in Chapter 6. For example, the `exists` method indicates whether or not a certain file exists, and the `isDirectory` method indicates whether or not the name supplied corresponds to a file or a directory.

Let's write a program that will prompt the user for the name of a file or directory and recursively explore all the files that can be reached from that starting point. If the user provides the name of a file, the program should simply print the name. But if the user gives the name of a directory, the program should print the directory name and list all the directories and files inside that directory.

We can write a fairly simple main method that prompts for a file or directory name and checks to make sure it exists. If it does not, the program tells the user that there is no such file or directory. If it does exist, the program calls a method to print the information for that `File` object:

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("directory or file name?");
    String name = console.nextLine();
    File f = new File(name);
    if (!f.exists()) {
        System.out.println("No such file/directory");
    } else {
        print(f);
    }
}
```

Our job will be to write the `print` method. The method should begin by printing the name of the file or directory:

```
public static void print(File f) {
    System.out.println(f.getName());
    ...
}
```

If the `File` object `f` represents a simple file, we have completed the task. But if `f` represents a directory, we also want to print the names of all the files contained in the directory:

```
public static void print(File f) {
    System.out.println(f.getName());
    if (f.isDirectory()) {
        // print contents of directory
    }
}
```

We can accomplish this task with the `listFiles` method, which returns an array of `File` objects that represent the contents of the directory. We can use a for-each loop to process each object:

```
public static void print(File f) {
    System.out.println(f.getName());
    if (f.isDirectory()) {
        for (File subF : f.listFiles()) {
            // print information for subF
        }
    }
}
```

To complete the method, we have to figure out how to print information about each of the individual `subF` objects in the directory. An obvious approach is to simply print the names of the subfiles:

```
// not quite right
public static void print(File f) {
    System.out.println(f.getName());
    if (f.isDirectory()) {
        for (File subF : f.listFiles()) {
            System.out.println(subF.getName());
        }
    }
}
```

This method works in that it prints the names of the subfiles inside the directory. But remember that there can be directories inside this directory, so some of those subfiles might actually be directories whose contents also need to be printed. We could try to fix our code by adding a test:

```
// getting worse, not better
public static void print(File f) {
    System.out.println(f.getName());
```



```

    if (f.isDirectory()) {
        for (File subF : f.listFiles()) {
            System.out.println(subF.getName());
            if (subF.isDirectory()) {
                // print contents of subdirectory
            }
        }
    }
}

```

But even this won't work, because there might be directories within those inner directories, and those directories might have subdirectories. There is no simple way to solve this problem with standard iterative techniques.

The solution is to think recursively. You might be tempted to envision many different cases: a file, a directory with files, a directory with subdirectories, a directory with subdirectories that have subdirectories, and so on. However, there are really only two cases to consider: Each object is either a file or a directory. If it's a file, we simply print its name. If it's a directory, we print its name and then print information about every file and directory inside of it. How do we get the code to recursively explore all of the possibilities? We call our own print method to process whatever appears inside a directory:

```

public static void print(File f) {
    System.out.println(f.getName());
    if (f.isDirectory()) {
        for (File subF : f.listFiles()) {
            print(subF);
        }
    }
}

```

This version of the code recursively explores the structure. Each time the method finds something inside a directory, it makes a recursive call that can handle either a file or a directory; that recursive call might make yet another recursive call to handle either a file or directory, and so on.

If we run this version of the method, we'll get output like the following:

```

homework
assignments.doc
hw1
Song.java
Song.class
hw2
DrawRocket.java
DrawRocket.class

```

The problem with this output is that it doesn't indicate the structure to us. We know that the first line of output is the name of the starting directory (homework) and that everything that follows is inside that directory, but we can't easily see the sub-structure. It would be more convenient if the output used indentation to indicate the inner structure, as in the following lines of output:

```
homework
  assignments.doc
  hw1
    Song.java
    Song.class
  hw2
    DrawRocket.java
    DrawRocket.class
```

In this output we can more clearly see that the directory called homework contains three elements, two of which are directories that have their own files (hw1 and hw2). We can get this output by including an extra parameter in the print method that indicates the desired level of indentation. On the initial call in main, we can pass the method an indentation of 0. On each recursive call, we can pass it a value one higher than the current level. We can then use that parameter to print some extra spacing at the beginning of the line to generate the indentation.

Here is our program, including the new version of the method with indentation:

```
1 // This program prompts the user for a file or directory name
2 // and shows a listing of all files and directories that can be
3 // reached from it (including subdirectories).
4
5 import java.io.*;
6 import java.util.*;
7
8 public class DirectoryCrawler {
9     public static void main(String[] args) {
10         Scanner console = new Scanner(System.in);
11         System.out.print("directory or file name?");
12         String name = console.nextLine();
13         File f = new File(name);
14         if (!f.exists()) {
15             System.out.println("No such file/directory");
16         } else {
17             print(f, 0);
18         }
19     }
20 }
```

```

21      // Prints information for the given file/directory using the
22      // given level of indentation
23      public static void print(File f, int level) {
24          for (int i = 0; i < level; i++) {
25              System.out.print("    ");
26          }
27          System.out.println(f.getName());
28          if (f.isDirectory()) {
29              for (File subF : f.listFiles()) {
30                  print(subF, level + 1);
31              }
32          }
33      }
34  }

```

Helper Methods

Sometimes when you solve a problem recursively, you need to introduce an extra method to solve the problem. We call such methods *helper* methods.

For example, consider the task of writing a method called `sum` that takes an array of integers as a parameter and that returns the sum of the numbers in the array. Even though this is a fairly simple task, it turns out to be challenging to implement it using recursion. We could make a base case for an empty array (an array of length 0):

```

public int sum(int[] list) {
    if (list.length == 0) {
        return 0;
    } else {
        ...
    }
}

```

If we write the code this way, what would we use for a recursive case? Suppose, for example, that we are asked to work with an array of length 10. How do we simplify an array of length 10 to bring it closer to being an empty array? The only way to do that would be to construct a new array with a shorter length, but that would be wasteful.

It is useful to consider how you would solve this **iteratively**. Here is an iterative version that follows a classic cumulative sum approach:

```

int sum = 0;
for (int i = 0; i < list.length; i++) {
    sum += list[i];
}

```

The key to the iterative approach is to work with an index variable that starts at 0 and continues iterating until it equals the length of the array. We can apply this same idea to a recursive solution. Remember that you want to think about how to pick off some small piece of the problem. Consider the following line of pseudocode:

```
sum(entire list) = list[0] + sum(list starting at 1)
```

This code states that we have to add in `list[0]`. That leaves us with the task of adding up the remaining values of the list (the values starting at index 1). And how would we do that? We can similarly say that:

```
sum(list starting at 1) = list[1] + sum(list starting at 2)
```

This approach leads to a series of recursive calls, each of which handles one element of the array:

```
sum(entire list) = list[0] + sum(list starting at 1)
    sum(list starting at 1) = list[1] + sum(list starting at 2)
        sum(list starting at 2) = list[2] + sum(list starting at 3)
            ...
```

When does this process end? We could end when we get to the last element of the array:

```
sum(list starting at list.length - 1) = list[list.length - 1]
```

Unfortunately, this choice of ending point assumes that a last element exists. That won't be true of an array of length 0. An easier approach is to end when we reach an index beyond the last element of the array. The `for` loop version of this method executes for as long as the index is less than the length of the array. Once the index becomes equal to the length of the array, it stops. Our recursion can similarly stop when the index becomes equal to the length of the array. And what does that add up to? It adds up to 0:

```
sum(list starting at list.length) = 0
```

Returning 0 in this case is the recursive equivalent of initializing the cumulative sum to 0.

We are almost ready to write the recursive method, but to express this idea as a method, we have to pass the method both the array and the starting index. So we want our method to look like the following:

```
// computes the sum of the list starting at the given index
public int sum(int[] list, int index) {
    ...
}
```


Using those two parameters, you'll find that the code is fairly simple to write using the approach described before:

```
// computes the sum of the list starting at the given index
public int sum(int[] list, int index) {
    if (index == list.length) {
        return 0;
    } else {
        return list[index] + sum(list, index + 1);
    }
}
```

Of course, there's the issue of that extra parameter. We were asked to write a method that is passed just the array. We could complain that we were asked to solve the wrong problem, but that's a bad attitude to have. We should allow a client to call methods using parameters that are convenient. We need the second parameter to write our recursive solution, but we shouldn't burden the client with the need to understand that detail.

This is a common situation. We often find ourselves wanting to write a recursive method that has extra parameters relative to the public method we are being asked to implement. The solution is to introduce a private method that has the parameter passing we want. So the client gets the convenient public method with the kind of parameter passing that the client wants, and we get our convenient private method with the kind of parameter passing that we want. We call the private method a helper method because it helps us to write the public method we've been asked to implement.

Here is the code that includes the private method:

```
// returns the sum of the numbers in the given array
public int sum(int[] list) {
    return sum(list, 0);
}

// computes the sum of the list starting at the given index
private int sum(int[] list, int index) {
    if (index == list.length) {
        return 0;
    } else {
        return list[index] + sum(list, index + 1);
    }
}
```

The public method simply calls the private method, passing it an appropriate value for the second parameter. Both methods can be called `sum` because they have

different signatures (one parameter versus two). We make the second method a private method because we don't want to clutter up the public interface seen by the client.

There is a final benefit to this two-method approach. Sometimes you want to perform an action once, either before or after the recursive method executes. By dividing the recursive method into a public and a private method, you can include this one-time code in the public method either just before or just after the call on the private method.

12.4 Recursive Graphics

There has been a great deal of interest in the past 30 years about an emerging field of mathematics called *fractal geometry*. A fractal is a geometric object that is recursively constructed or self-similar. A fractal shape contains smaller versions of itself, so that it looks similar at all magnifications.

Benoit Mandelbrot founded the field of fractals in 1975 with his first publication about these intriguing objects, particularly a specific fractal that has come to be known as the Mandelbrot set. The most impressive aspect of fractal geometry is that extremely intricate and complex phenomena can be described with a simple set of rules. When Mandelbrot and others began drawing pictures of their fractals, they were an instant hit.

Many fractals can be described easily with recursion. As an example, we will explore a recursive method for drawing what is known as the Sierpinski triangle. We can't draw the actual fractal, because it is composed of infinitely many subtriangles. Instead, we will write a method that produces various levels that approximate the actual fractal.

At level 1, we draw an equilateral triangle as shown in Figure 12.1:

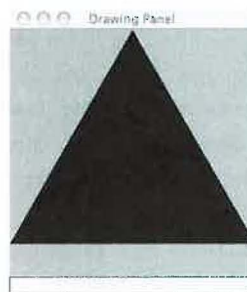


Figure 12.1 Sierpinski triangle, level 1

Proceeding to level 2, we draw three smaller triangles that are contained within the original triangle (see Figure 12.2):

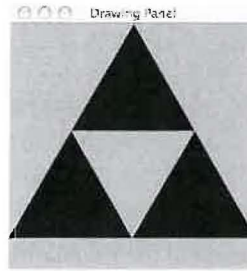


Figure 12.2 Sierpinski triangle, level 2

We apply this principle in a recursive manner. Just as we replaced the original triangle with three inner triangles, we replace each of these three triangles with three inner triangles to obtain Figure 12.3 with nine triangles in level 3:

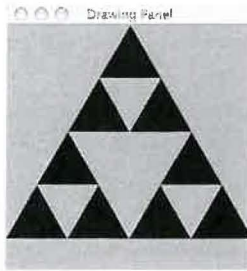


Figure 12.3 Sierpinski triangle, level 3

This process continues indefinitely, making a more intricate pattern at each new level. Figure 12.4 shows the result at level 7:

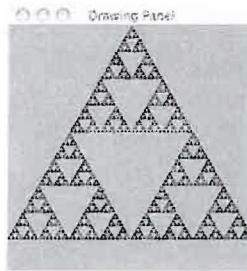


Figure 12.4 Sierpinski triangle, level 7

We can solve this problem using the `DrawingPanel` class from Supplement 3G. We'll pass the `Graphics` object for the panel to the method that is to draw the

triangles. The method will also need to know the level to use and the three vertices of the triangle, which we can pass as `Point` objects. Our method will look like this:

```
public static void drawFigure(int level, Graphics g,
                             Point p1, Point p2, Point p3) {
    ...
}
```

Our base case will be to draw the basic triangle for level 1. The `Graphics` class has methods for filling rectangles and ovals, but not for filling triangles. Fortunately, there is a `Polygon` class in the `java.awt` package. We can construct a `Polygon` and then add a series of points to it. Once we've specified the three vertices, we can use the `fillPolygon` method of the `Graphics` class to fill the polygon. Our base case will look like this:

```
public static void drawFigure(int level, Graphics g,
                             Point p1, Point p2, Point p3) {
    if (level == 1) {
        // base case: simple triangle
        Polygon p = new Polygon();
        p.addPoint(p1.x, p1.y);
        p.addPoint(p2.x, p2.y);
        p.addPoint(p3.x, p3.y);
        g.fillPolygon(p);
    } else {
        // recursive case, split into 3 triangles
        ...
    }
}
```

Most of the work happens in the recursive case. We have to split the triangle into three smaller triangles. We'll label the vertices of the overall triangle (see Figure 12.5).

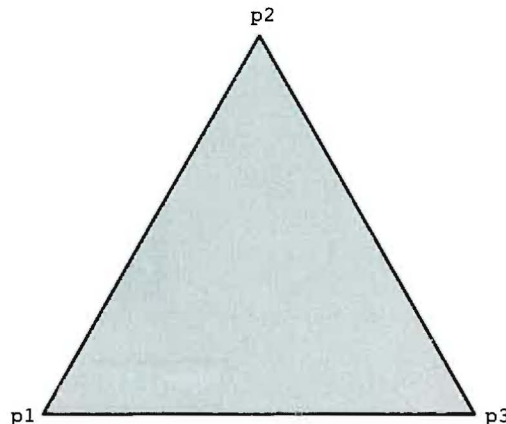


Figure 12.5 Triangle before splitting

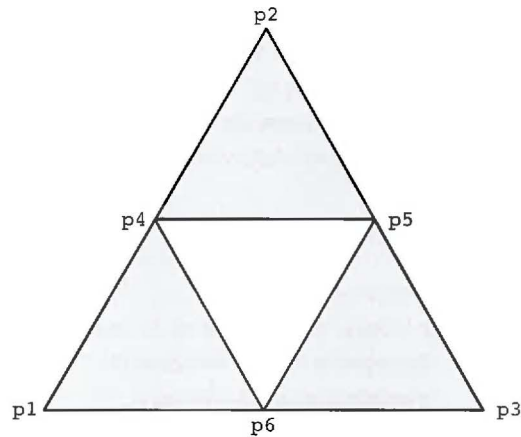


Figure 12.6 Triangle split into subtriangles

We then need to compute three new points that are the midpoints of the three sides of this triangle as shown in Figure 12.6.

From Figure 12.6, we see that

- p4 is the midpoint of p1 and p2
- p5 is the midpoint of p2 and p3
- p6 is the midpoint of p1 and p3

Once we have computed those points, we can describe the smaller triangles as follows:

- In the lower-left corner is the triangle formed by p1, p4, and p6.
- On top is the triangle formed by p4, p2, and p5.
- In the lower-right corner is the triangle formed by p6, p5, and p3.

There are three different midpoint computations involved here, so clearly it will be helpful to first write a method that will compute the midpoint of a segment given two endpoints. Computing the midpoints involves finding the arithmetic average (halfway point) of the x values and the y values:

```
public static Point midpoint(Point p1, Point p2) {  
    return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);  
}
```

Given this method, we can easily compute the three midpoints.

The final detail we have to think about is the level. If you look again at the level 2 version of the figure (Figure 12.2), you will notice that it is composed of three simple

triangles. In other words, the level 2 figure is composed of three level 1 figures. Similarly, the level 3 figure is composed of three level 2 figures, each of which in turn is composed of three level 1 figures. In general, if we are asked to draw a level n figure, we do so by drawing three level $(n - 1)$ figures.

We can turn these observations into code to complete the recursive method:

```
public static void drawFigure(int level, Graphics g,
                             Point p1, Point p2, Point p3) {
    if (level == 1) {
        // base case: simple triangle
        Polygon p = new Polygon();
        p.addPoint(p1.x, p1.y);
        p.addPoint(p2.x, p2.y);
        p.addPoint(p3.x, p3.y);
        g.fillPolygon(p);
    } else {
        // recursive case, split into 3 triangles
        Point p4 = midpoint(p1, p2);
        Point p5 = midpoint(p2, p3);
        Point p6 = midpoint(p1, p3);

        // recurse on 3 triangular areas
        drawFigure(level - 1, g, p1, p4, p6);
        drawFigure(level - 1, g, p4, p2, p5);
        drawFigure(level - 1, g, p6, p5, p3);
    }
}
```

There is a limit to the number of levels down that we can go. The `DrawingPanel` has a finite resolution, so at some point we won't be able to subdivide our triangles any further. Also bear in mind that at each new level the number of triangles that we draw triples, which means that the number of triangles increases exponentially with the level.

The complete program is available from <http://www.buildingjavaprograms.com>. Remember that you need to also download `DrawingPanel.java` to execute the program.

12.5 Recursive Backtracking

Many programming problems can be solved by systematically searching a set of possibilities. For example, if you want to find a path through a maze from a starting point to an exit point, you can explore all possible paths through the maze until you find one that works. For many games like tic-tac-toe you can explore all possible moves and countermoves to see if there is some move that guarantees that you win.

Many of these exhaustive search problems can be solved with an approach called *backtracking*. It is a particular approach to problem solving that is nicely expressed using recursion. As a result, it is sometimes referred to as *recursive backtracking*.

(Recursive) Backtracking

A general algorithm for finding solutions to a problem by exploring possible candidate solutions and abandoning (“backtracking”) once a given candidate is deemed unsuitable.

Backtracking involves searching all possibilities, so it can be an expensive technique to use. But many problems are small enough in scope that they are nicely solved with a backtracking approach.

A Simple Example: Traveling North/East

To introduce the basic concepts and terminology of backtracking, let’s explore a simple example. Consider a standard Cartesian plane with (x, y) coordinates. Suppose that you start at the origin, $(0, 0)$, and you are allowed to repeatedly make one of these three moves:

- You can move North (abbreviated “N”), which will increase your y -coordinate by 1.
- You can move East (abbreviated “E”), which will increase your x -coordinate by 1.
- You can move Northeast (abbreviated “NE”), which will increase both your x -coordinate and y -coordinate by 1.

Starting from the origin, these three different moves would leave you in the locations shown in Figure 12.7. We can think of this as a traveling problem where we can make a series of moves that take us from the origin to some other (x, y) point. For example, the sequence of moves N, NE, N would leave us at $(1, 3)$.

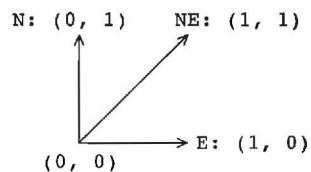


Figure 12.7 Traveling north, east, or northeast from the origin

Every backtracking problem involves a *solution space* of possible answers that you want to explore. We try to view the problem as a sequence of *choices*, which allows us to think of the solution space as a *decision tree*. For our traveling problem the choices are the sequence of moves that we make. Figure 12.8 shows a decision tree showing all of the possible ways to make two moves and where each sequence leaves us. These decision trees can be quite large even for a small problem like this.

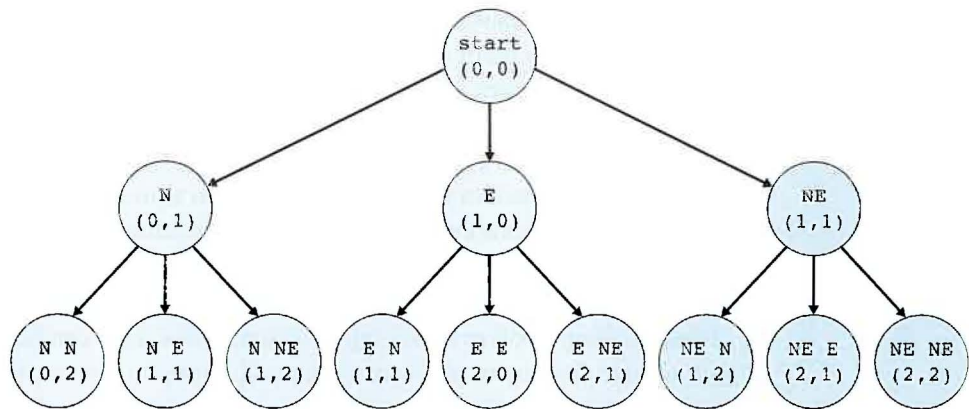


Figure 12.8 Decision tree for two moves

Consider the problem of traveling from the origin to the point (1, 2). What are the possible sequences of moves that would get you there? People are fairly good at solving problems like these, so you can probably pretty easily come up with all five of the possibilities:

- N, N, E
- N, E, N
- N, NE
- E, N, N
- NE, N

How would we write a computer program to find these solutions? For this simple problem, we could devise a specialized algorithm that takes into account the properties of these paths, but backtracking provides a convenient way to exhaustively search all possibilities.

For most backtracking solutions we end up writing two methods. We generally want a public method that is passed the details of the problem to be solved. But we almost always need extra parameters for the backtracking, so we almost always have an extra private method that does the actual backtracking.

The basic form that our backtracking solution will take is a method to explore all possible choices recursively:

```
private static void explore(a scenario) {
    ...
}
```


Because we are using recursion, we need to identify base cases and recursive cases. Backtracking solutions generally involve two different base cases. You tend to stop the backtracking when you find a solution, so that becomes one of our base cases. Often what you want to do when you find a solution is to report it:

```
private static void explore(a scenario) {
    if (this is a solution) {
        report it.
    } else {
        ...
    }
}
```

We don't want to search forever, so we also have to be on the lookout for what is called a *dead-end*. We might make a set of choices that lead us to a scenario where it is clear that no solution is possible with this set of choices. We make this a base case in our recursion so that we stop exploring when we reach a dead-end:

```
private static void explore(a scenario) {
    if (this is a solution) {
        report it.
    } else if (this is not a dead-end) {
        ...
    }
}
```

If our backtracking search has led us to a scenario where we haven't yet solved the problem and haven't yet reached a dead-end, then we want to explore each possible choice available to us. For each possible choice, we recursively explore the scenario of making that choice:

```
private static void explore(a scenario) {
    if (this is a solution) {
        report it.
    } else if (this is not a dead-end) {
        use recursive calls to explore each available choice.
    }
}
```

This pseudocode captures the essence of the backtracking approach. Not all backtracking solutions will take exactly this form, but they will all have some variation of each of these elements.

We can flesh out the pseudocode a bit by filling in some details for this particular problem. We are considering the problem of moving from a current position to a

target position, both specified with (x, y) coordinates. And we have three available moves from any given position: N, E, and NE:

```
private static void explore(current (x, y) and target (x, y)) {
    if (this is a solution) {
        report it.
    } else if (this is not a dead-end) {
        explore(moving N).
        explore(moving E).
        explore(moving NE).
    }
}
```

Often it can be challenging to figure out what parameters to pass to the exploration method. In this case, we need a current x and y and a target x and y . We also need some way of keeping track of the choices that we have made so that we can report the path that we have taken. There are many ways to do this, but for simplicity, let's build up a string that stores the sequence of moves.

We can test whether we have a solution by testing whether the current and target coordinates match. But how do we test for a dead-end? In this problem, our x -coordinates and y -coordinates never go down. If we reach a point where the current x is greater than the target x or the current y is greater than the target y , then we know that we have traveled too far in that direction and we will never reach the target.

Putting these details together, we end up with the following exploration method:

```
private static void explore(int targetX, int targetY,
                           int currX, int currY, String path) {
    if (currX == targetX && currY == targetY) {
        System.out.println(path);
    } else if (currX <= targetX && currY <= targetY) {
        explore(targetX, targetY, currX, currY + 1, path + " N");
        explore(targetX, targetY, currX + 1, currY, path + " E");
        explore(targetX, targetY, currX + 1, currY + 1, path + " NE");
    }
}
```

We can complete the solution by providing a public method that starts the recursive search by passing the initial coordinates $(0, 0)$ and an appropriate value to appear at the front of the string that stores the path:

```
public static void travel(int targetX, int targetY) {
    explore(targetX, targetY, 0, 0, "moves:");
}
```

If we ask the method to find all ways of traveling to (1, 2), it produces the following output:

```
moves: N N E
moves: N E N
moves: N NE
moves: E N N
moves: NE N
```

Figure 12.9 shows the complete decision tree that is explored by this particular call with the five solutions shaded.

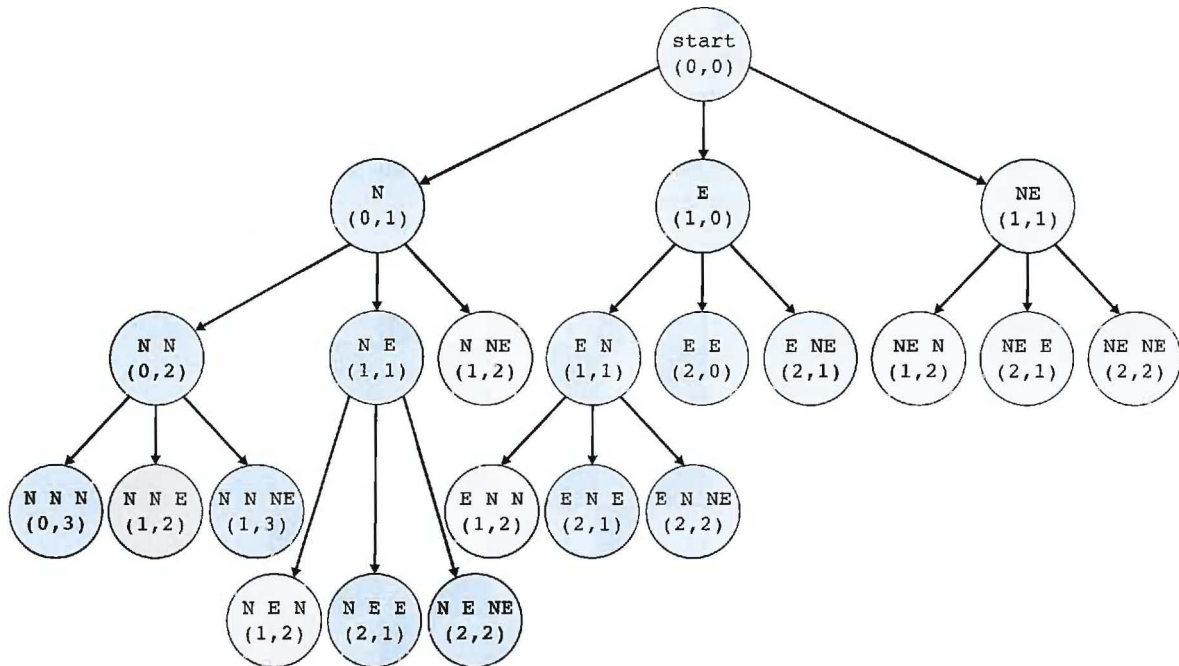


Figure 12.9 Complete decision tree for paths to (1, 2)

Notice that the tree does not contain all of the possible combinations of three moves. That is because we don't explore past a solution and we don't explore past a dead-end (such as (2, 1)).

It is useful to consider the order in which the recursive method searches these possibilities. In general, as with most recursive solutions, the recursive calls lead to a depth-first traversal. From the starting point at the top of the tree it first considers moving North to (0, 1). That entire branch of the tree is explored before any other branches are explored. The next call considers moving North again to (0, 2). Then it

considers another move North to (0, 3). So the initial sequence of calls leads it down to the leftmost node of the three. This is a dead-end, so it stops exploring. At that point, it returns to where it last had another choice to consider. It backs up to where it had chosen two North moves in a row (N N) and it considers moving East instead of moving North. That turns out to be a solution (N N E), so it reports a success. Then it backs up again to where it last had another choice to make and it considers the remaining possibilities.

This property of going back to where it still has other choices to explore is the source of the term “backtracking.” As it finds solutions and dead-ends, it goes back (backtracks) to where it last had some other move to consider. It continues to search until it has exhausted all possible sequences of choices.

8 Queens Puzzle

A classic puzzle that backtracking solves well is to find a way to place 8 queens on a chessboard so that no two queens threaten each other. Because queens can move horizontally, vertically, or diagonally, it is a challenge to find a way to put 8 different queens on the board so that no two of them are in the same row, column, or diagonal. Figure 12.10 shows one example placement.

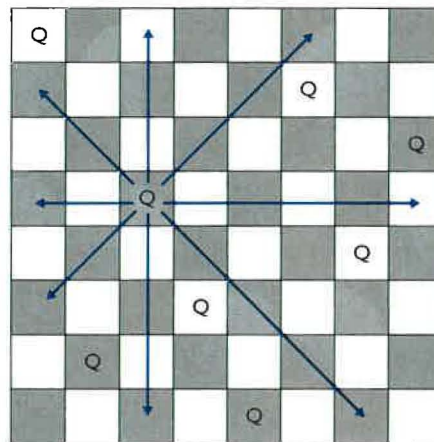


Figure 12.10 The 8 queens problem

To solve this problem with backtracking, we have to think of it in terms of a sequence of choices. The simplest way to do this is to think in terms of choosing where the first queen goes, then choosing where the second queen goes, and so on. There are 64 places to put the first queen because the chessboard is an 8-by-8 board. At the top of the tree, there are 64 different choices you could make for placing the first queen. Then once you've placed one queen, there are 63 squares left to choose from for the second queen, then 62 squares for the third queen, and so on.

Because backtracking searches all possibilities, it can take a long time to execute as there are many possibilities to explore. If we explore them all, we'll need to look at $64 * 63 * 62 * \dots * 57$ states, which is too many even for a fast computer. We need to be as smart as we can about the choices we explore. In the case of 8 queens, we can do better than to consider 64 choices followed by 63 choices followed by 62 choices, and so on. We know that most of these aren't worth exploring.

One approach is to observe that if there is any solution at all to this problem, then the solution will have exactly one queen in each row and exactly one queen in each column. That's because you can't have two in the same row or two in the same column and there are 8 of them on an 8-by-8 board. We can search more efficiently if we go row-by-row or column-by-column. It doesn't matter which choice we make, so let's explore column by column. Eliminating undesirable candidates from being explored is also called *pruning* the decision tree.

In this new way of looking at the search space, the first choice is for column 1. We have 8 different rows where we could put a queen in column 1. At the next level we consider all of the places to put a queen in column 2, and so on. Figure 12.11 shows the top of our decision tree.

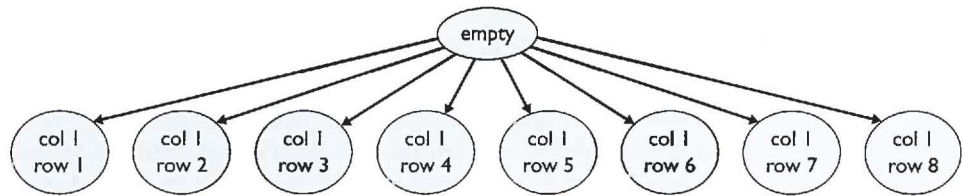


Figure 12.11 Decision tree for first column

There are 8 different branches for each column. Under each of these branches, we have 8 branches for each of the possible rows where we might place a queen in column 2. For example, if we think just about the possibility of placing the first queen in column 1 and row 5 and then think about all of the ways to place a second queen, we end up with an extra level of the tree as shown in Figure 12.12.

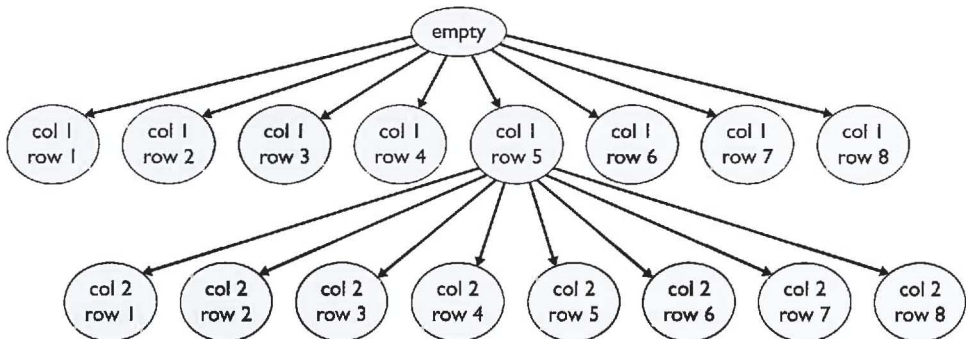


Figure 12.12 A decision tree for second column

These pictures don't capture the whole story because the tree is so large. There are 8 branches at the top. From each of these 8 branches there are 8 branches. And from each of those branches there are 8 branches. This continues eight levels deep (one level for each column of the board).

It's clear that the 8 choices could be coded fairly nicely in a `for` loop, something along the lines of:

```
for (int row = 1; row <= 8; row++) {
```

But what we need for backtracking is something more like a deeply nested `for` loop:

```
for (int row = 1; row <= 8; row++) { // explore column 1
    for (int row = 1; row <= 8; row++) { // explore column 2
        for (int row = 1; row <= 8; row++) { // explore column 3
            for (int row = 1; row <= 8; row++) { // explore column 4
                ....
            }
        }
    }
}
```

That's not a bad way to think of what backtracking does, although we will use recursion to write this in a more elegant way.

Before we explore the backtracking code, we need to consider the low-level details of how to keep track of a board that allows us to place queens in specific locations. It is helpful to split off the low-level details into a separate class. Let's plan on writing a `Board` class that allows us to construct a `Board` object to keep track of the state of the chessboard.

What kind of methods would we want to have for a `Board` object? Obviously it would need some kind of constructor. We want to pass it an integer n so we could solve the more general " n queens" problem with an n -by- n board. We need to be able to test whether it's safe to place a queen at a particular location. We need a way to place a queen on the board. We need a way to remove a queen because the backtracking involves trying different possibilities. We need a way to display output showing where the queens have been placed. Lastly, we'd like to be able to ask the board what size it is. These can be implemented as the following constructor and methods:

```
public Board(int size)
public boolean isSafe(int row, int column)
public void place(int row, int column)
public void remove(int row, int column)
public void print()
public int size()
```

Let's assume that we have a `Board` class that implements all of these methods. That's not the interesting code. The interesting code is the backtracking code which, given this class, we can now write in a very straightforward manner.

Once again we will introduce a public/private pair of methods to perform the backtracking. The public method can be passed a `Board` object and it will call the private recursive method that performs the backtracking:

```
public static void solve(Board b) {
    ...
}
```

Recall that we had this general pseudocode for backtracking solutions:

```
private static void explore(a scenario) {
    if (this is a solution) {
        report it.
    } else if (this is not a dead-end) {
        use recursive calls to explore each available choice.
    }
}
```

The 8-queens backtracking problem differs in several important ways from the simple backtracking we saw before. Let's consider each of the differences and see how to adapt our pseudocode for each of them.

In the traveling problem we had just three possibilities to consider, so it made sense to write three recursive calls to explore each possibility. Here we have eight possibilities for the 8-queens problem and potentially a different number of possibilities if the board size is something other than 8. In this case we want to use a loop to consider the different possibilities. Many backtracking problems will require using a loop instead of individual calls; it is useful to adapt our pseudocode to fit that approach.

```
private static void explore(a scenario) {
    if (this is a solution) {
        report it.
    } else {
        for (each available choice) {
            if (this is not a dead-end) {
                use a recursive call to explore that choice.
            }
        }
    }
}
```

We can also be more specific about what it means to explore a choice. In our simple example, we were building up a string that stored the path. That meant that we didn't have to undo a choice to move on to the next choice. More complex backtracking prob-

lems require a cleanup step where you undo a choice. That will be true of 8-queens. We will place a queen on the board to explore that branch and when we come back from the recursive exploration, we will need to remove the queen to get ready to explore the next possible choice. So the pseudocode can be expanded to include the pattern of making a choice, recursively exploring, and then undoing the choice:

```
private static void explore(a scenario) {
    if (this is a solution) {
        report it.
    } else {
        for (each available choice) {
            if (this is not a dead-end) {
                make the choice.
                recursively explore subsequent choices.
                undo the choice.
            }
        }
    }
}
```

This pseudocode is general enough that it can be used for many backtracking problems. Some programmers refer to this as the “choose, explore, un-choose” pattern for backtracking.

This is appropriate code to use if you want to search all possibilities. In the case of 8-queens, there are many solutions (over 90 different solutions). We don’t really want to see all of these solutions. We’re happy to have just one. For this backtracking problem, we will consider a variation that stops when it finds a solution. That means that our recursive method will need to have a way to let us know whether a certain path worked out or whether it turned out to be a dead-end. A good way to do this is to have a `boolean` return type and to have the method return `true` if it succeeds, `false` if it is a dead-end.

```
private static boolean explore(a scenario) {
    if (this is a solution) {
        report it.
        return true;
    } else {
        for (each available choice) {
            if (this is not a dead-end) {
                make the choice.
                if (recursive call to explore subsequent choices) {
                    return true;
                }
            }
        }
    }
}
```

```

        undo the choice.
    }
}
return false;
}

```

This pseudocode is also general enough that it can be used for many backtracking problems when you want to stop the process after finding a solution.

We can adapt this to the 8-queens problem by filling in the details. What parameters will it need to specify a scenario? It certainly needs the `Board` object. Recall that each level of the decision tree involves a different column of the board. The first invocation will handle column 1, the second will handle column 2, and so on. Therefore, in addition to the board, the method also needs to know the column to work on. That leaves us with this header:

```

private static boolean explore(Board b, int col) {
    ...
}

```

How do we know if we've found a solution? This backtracking code doesn't explore dead-ends. As a result, it has the following precondition:

```
// pre: queens have been safely placed in previous columns
```

What would be a nice column to get to? A tempting answer is 8. It would be nice to get to column 8 because it would mean that 7 of the 8 queens have been placed properly. But an even better answer is 9, because the precondition tells us that if we ever reach column 9, then queens have been safely placed in each of the first 8 columns.

This turns out to be our test for whether we have found a solution:

```

private static boolean explore(Board b, int col) {
    if (col > b.size()) {
        return true;
    } else {
        ...
    }
}

```

The pseudocode indicates that we should print the answer before returning `true`. We could do that, but because the solution is stored in the `Board` object, we can simply return and allow the calling method to print out the solution.

Now we have to fill in the details of the `for` loop that explores the various possibilities. We have 8 possibilities to explore (the 8 rows of this column where we might place a queen). A `for` loop works nicely to explore the different row numbers. The

pseudocode indicates that we should test to make sure it is not a dead-end. We can do that by making sure that it is safe to place a queen in that row and column:

```
for (int row = 1; row <= b.size(); row++) {
    if (b.isSafe(row, col)) {
        ...
    }
}
```

We now need to fill in the three steps that are involved in exploring a choice: making the choice, recursively exploring subsequent choices, and undoing the choice. We make the choice by telling the board to place a queen in that row and column:

```
for (int row = 1; row <= b.size(); row++) {
    if (b.isSafe(row, col)) {
        b.place(row, col);
        ...
    }
}
```

Then we recursively explore subsequent choices (later columns) and return true to stop the process if it finds a solution:

```
for (int row = 1; row <= b.size(); row++) {
    if (b.isSafe(row, col)) {
        b.place(row, col);
        if (explore(b, col + 1)) {
            return true;
        }
        ...
    }
}
```

Finally we have to undo our choice in case that turns out to be a dead-end:

```
for (int row = 1; row <= b.size(); row++) {
    if (b.isSafe(row, col)) {
        b.place(row, col);
        if (explore(b, col + 1)) {
            return true;
        }
        b.remove(row, col);
    }
}
```

Putting it all together we get:

```
private static boolean explore(Board b, int col) {
    if (col > b.size()) {
        return true;
    } else {
        for (int row = 1; row <= b.size(); row++) {
            if (b.isSafe(row, col)) {
                b.place(row, col);
                if (explore(b, col + 1)) {
                    return true;
                }
                b.remove(row, col);
            }
        }
        return false;
    }
}
```

We need some code in the solve method that starts the recursion in column 1 and that either prints the solution or a message about there not being solutions:

```
public static void solve(Board solution) {
    if (explore(solution, 1)) {
        System.out.println("One solution is as follows:");
        solution.print();
    } else {
        System.out.println("No solution.");
    }
}
```

On our web site <http://www.buildingjavaprograms.com/> you will find the Board class and a full runnable version of the program that you can run to see that it finds a solution to 8 queens, 4 queens, and so on. In the case of 2 queens, there are no solutions. Our graphical variation of the program includes an animation generated by the backtracking itself.

Solving Sudoku Puzzles

Let's look at one more example of backtracking. Sudoku puzzles have become very popular. The puzzle involves a 9-by-9 grid that is to be filled in with the digits 1 through 9. Each row and column is required to have exactly one occurrence of each of the nine digits. The grid is further divided into nine 3-by-3 grids, and each grid is also required to have exactly one occurrence of each of the nine digits. A specific Sudoku puzzle will fill in some of the cells of the grid with specific digits, such as the puzzle shown in Figure 12.13.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

Figure 12.13 A Sudoku board

When people play Sudoku, they use all sorts of heuristics and intuitions to find a way to fill in the empty cells of the grid while still meeting the constraints. With a backtracking approach, we just use brute force to try all possibilities.

The choices involved are what digit to put in each of the unoccupied cells of the grid. There are nine possibilities to explore for each empty cell (the digits 1 through 9). This approach to Sudoku is not necessarily going to run quickly, but backtracking provides a framework for quickly writing a complete solution.

As with many backtracking tasks, the real work involves keeping track of the details of this particular scenario. With the 8-queens problem we introduced a `Board` class to keep track of the chessboard. In this case we should develop a `Grid` class that keeps track of the state of the Sudoku grid.

In the case of the Sudoku grid, we have to start with some of the cells filled in. That will mean that our `Grid` class will need to read the initial configuration. Let's assume that each puzzle is stored in a text file with a 9-by-9 grid of numbers and dashes, as in:

```

3 - 6 5 - 8 4 - -
5 2 - - - - -
- 8 7 - - - 3 1
- - 3 - 1 - - 8 -
9 - - 8 6 3 - - 5
- 5 - - 9 - 6 - -
1 3 - - - - 2 5 -
- - - - - - 7 4
- - 5 2 - 6 3 - -

```

That means that our Grid class constructor will need to read the initial configuration:

```
public Grid(Scanner input)
```

We immediately run into another problem. If some of the cells are filled in, then it's not as simple as the 8-queens case where we could just systematically explore each different cell of an empty board. In the Sudoku case, we have to figure out which cell to explore first, which cell to go to after that, and so on. There are many ways we could approach this, but the simplest is to give this responsibility to the Grid class. We can introduce a method that allows us to ask the grid to give us the next unassigned location in the grid.

Then we are faced with the question of how to specify a grid location. An obvious approach would be to give row and column numbers, but a method can return only one value. If the grid is going to identify the unassigned locations, then we don't really need to keep track of rows and columns ourselves. To keep thing simple, let's assume that the grid returns a cell number as a simple int:

```
public int getUnassignedLocation()
```

Eventually the grid will fill up, so we have to adopt a convention for what happens when there is no unassigned location left. Let's assume that the method returns -1 in that case to indicate that the grid is full.

In addition to these methods, we will want similar methods to the ones we had in 8-queens. We want to be able to test whether it's safe to set a particular cell to a particular digit. This is the method that will make sure that the grid we are building up is legal. So it will have to check to see if this new digit doesn't already appear in the given row, column, or subgrid. We'll need a method to set a cell to a specific digit. An "undo" method to remove a digit from a cell will be necessary. We will also want a method to print our solution. These are implemented as the following methods, respectively:

```
public boolean noConflicts(int cellNumber, int n)
public void place(int cellNumber, int n)
public void remove(int cellNumber)
public void print()
```

As with 8-queens, we'll want to stop searching once we find a solution, so we start with the standard pseudocode for the single-solution backtracking:

```
private static boolean explore(a scenario) {
    if (this is a solution) {
        report it.
        return true;
    } else {
        for (each available choice) {
            if (this is not a dead-end) {
```



```

        make the choice.
        if (recursive call to explore subsequent choices) {
            return true;
        }
        undo the choice.
    }
}
return false;
}
}

```

In this case, the information about a scenario is stored in the grid object. We know that we have a solution if we manage to fill every cell of the grid. To test whether we have a solution we can ask the grid to give us the next unassigned location and see if the method returns -1 (the value that signals that the grid is full!):

```

private static boolean explore(Grid g) {
    int cellNumber = g.getUnassignedLocation();
    if (cellNumber == -1) {
        return true;
    } else {
        ...
    }
}

```

The available choices are the digits 1 through 9 and we have methods for performing each of the operations described in the pseudocode, so we can fill in the blanks to complete the method as follows:

```

private static boolean explore(Grid g) {
    int cellNumber = g.getUnassignedLocation();
    if (cellNumber == -1) {
        return true;
    } else {
        for (int n = 1; n <= 9; n++) {
            if (g.noConflicts(cellNumber, n)) {
                g.place(cellNumber, n);
                if (explore(g)) {
                    return true;
                }
                g.remove(cellNumber);
            }
        }
        return false;
    }
}

```

You will find the `Grid` class and a full runnable version of this program on our web site <http://www.buildingjavaprograms.com/> along with a graphical version that shows the choices as they are being considered by the backtracking.

12.6 Case Study: Prefix Evaluator



In this section, we will explore the use of recursion to evaluate complex numeric expressions. First we will examine the different conventions for specifying numeric expressions and then we will see how recursion makes it relatively easy to implement one of the standard conventions.

Infix, Prefix, and Postfix Notation

When we write numeric expressions in a Java program, we typically put numeric operators like `+` and `*` between the two operands, as in the following:

```
3.5 + 8.2
9.1 * 12.7
7.8 * (2.3 + 2.5)
```

Putting the operator between the operands is a convention known as *infix notation*. A second convention is to put the operator in front of the two operands, as in the following examples:

```
+ 3.5 8.2
* 9.1 12.7
* 7.8 + 2.3 2.5
```

Putting the operator in front of the operands is a convention known as *prefix notation*. Prefix notation looks odd for symbols like `+` and `*`, but it resembles mathematical function notation, in which the name of the function goes first. For example, if we were calling methods instead of using operators, we would write

```
plus(3.5, 8.2)
times(9.1, 12.7)
times(7.8, plus(2.3, 2.5))
```

There is also a third convention, in which the operator appears after the two operands, as in the following examples:

```
3.5 8.2 +
9.1 12.7 *
7.8 2.3 2.5 + *
```

This convention is known as *postfix notation*. It is also sometimes referred to as reverse Polish notation, or RPN. For many years Hewlett-Packard has sold scientific calculators that use RPN rather than normal infix notation.

We are so used to infix notation that it takes a while to get used to the other two conventions. One of the interesting facts you will discover if you take the time to learn the prefix and postfix conventions is that infix is the only notation that requires parentheses. The other two notations are unambiguous.

Table 12.1 summarizes the three notations.

Table 12.1 Arithmetic Notations

Notation	Description	Examples
infix	operator between operands	2.3 + 4.7 2.6 * 3.7 (3.4 + 7.9) * 18.6 + 2.3 / 4.7
prefix	operator before operands (functional notation)	+ 2.3 4.7 * 2.6 3.7
postfix	operator after operands (reverse Polish notation)	+ * + 3.4 7.9 18.6 / 2.3 4.7 2.3 4.7 + 2.6 3.7 * 3.4 7.9 + 18.6 * 2.3 4.7 / +

Evaluating Prefix Expressions

Of the three standard notations, prefix notation is most easily implemented with recursion. In this section we will write a method that reads a prefix expression from a `Scanner` and computes its value. Our method will look like this:

```
// pre : input contains a legal prefix expression
// post: expression is consumed and the result is returned
public static double evaluate(Scanner input) {
    ...
}
```

Before we can begin writing the method, we have to consider the kind of input that we are going to get. As the precondition indicates, we will assume that the `Scanner` contains a legal prefix expression. The simplest possible expression would be a number like

38.9

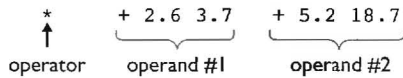
There isn't much to evaluate in this case—we can simply read and return the number. More complex prefix expressions will involve one or more operators. Remember that the operator goes in front of the operands in a prefix expression. A slightly more complex expression would be two numbers as operands with an operator in front:

+ 2.6 3.7

This expression could itself be an operand in a larger expression. For example, we might ask for

* + 2.6 3.7 + 5.2 18.7

At the outermost level, we have a multiplication operator with two operands:



In other words, this expression is computing the product of two sums. Here is the same expression written in the more familiar infix notation:

$(2.6 + 3.7) * (5.2 + 18.7)$

These expressions can become arbitrarily complex. The key observation to make about them is that they all begin with an operator. In other words, every prefix expression is of one of two forms:

- A simple number
- An operator followed by two operands

This observation will become a road map for our recursive solution. The simplest prefix expression will be a number, and we can distinguish it from the other case because any other expression will begin with an operator. So we can begin our recursive solution by checking whether the next token in the `Scanner` is a number. If it is, we have a simple case and we can simply read and return the number:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        ...
    }
}
```

Turning our attention to the recursive case, we know that the input must be an operator followed by two operands. We can begin by reading the operator:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        ...
    }
}
```


At this point we reach a critical decision. We have read in the operator, and now we need to read in the first operand and then the second operand. If we knew that the operands were simple numbers, we could write code like the following:

```
// not the right approach
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        double operand1 = input.nextDouble();
        double operand2 = input.nextDouble();
        ...
    }
}
```

But we have no guarantee that the operands are simple numbers. They might be complex expressions that begin with operators. Your instinct might be to test whether or not the original operator is followed by another operator (in other words, whether the first operand begins with an operator), but that reasoning won't lead you to a satisfactory outcome. Remember that the expressions can be arbitrarily complex, so either of the operands might contain dozens of operators to be processed.

The solution to this puzzle involves recursion. We need to read two operands from the scanner, and they might be very complex. But we know that they are in prefix form and we know that they aren't as complex as the original expression we were asked to evaluate. The key is to recursively evaluate each of the two operands:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        double operand1 = evaluate(input);
        double operand2 = evaluate(input);
        ...
    }
}
```

This simple solution works. Of course, we still have the task of evaluating the operator. After the two recursive calls have executed, we will have an operator and

two numbers (say, +, 3.4, and 2.6). It would be nice if we could just write a statement like the following:

 `return operand1 operator operand2; // does not work`

Unfortunately, Java doesn't work that way. We have to use a nested `if/else` statement to test what kind of operator we have and to return an appropriate value:

```
if (operator.equals("+")) {
    return operand1 + operand2;
} else if (operator.equals("-")) {
    return operand1 - operand2;
} else if (operator.equals("*")) {
    ...
}
```

We can include this code in its own method so that our recursive method stays fairly short:

```
public static double evaluate(Scanner input) {
    if (input.hasNextDouble()) {
        // base case with a simple number
        return input.nextDouble();
    } else {
        // recursive case with an operator and two operands
        String operator = input.next();
        double operand1 = evaluate(input);
        double operand2 = evaluate(input);
        return apply(operator, operand1, operand2);
    }
}
```

Complete Program

When you program with recursion, you'll notice two things. First, the recursive code that you write will tend to be fairly short, even though it might be solving a very complex task. Second, most of your program will generally end up being supporting code for the recursion that does low-level tasks. For our current task of evaluating a prefix expression, we have a short and powerful prefix evaluator, but we need to include some supporting code that explains the program to the user, prompts for a prefix expression, and reports the result. We also found that we needed a method that would apply an operator to two operands. The nonrecursive parts of the program are fairly straightforward, so they are included in the following code without detailed discussion:

```
1 // This program prompts for and evaluates a prefix expression.
2
```



```

47         }
48     }
49 }

```

The program can handle simple numbers, as in the following sample execution:

```

This program evaluates prefix
expressions that include the
operators +, -, *, / and %
expression? 38.9
value = 38.9

```

It can also handle expressions with a single operator, as in the following execution:

```

This program evaluates prefix
expressions that include the
operators +, -, *, / and %
expression? + 2.6 3.7
value = 6.300000000000001

```

And it handles the case we considered that involved a product of two sums:

```

This program evaluates prefix
expressions that include the
operators +, -, *, / and %
expression? * + 2.6 3.7 + 5.2 18.7
value = 150.57000000000002

```

In fact, it can handle arbitrarily complex expressions, as in the following sample execution:

```

This program evaluates prefix
expressions that include the
operators +, -, *, / and %
expression? / + * - 17.4 8.9 - 3.9 4.7 18.4 - 3.8 * 7.9 2.3
value = -0.8072372999304106

```

The expression being computed in the previous example is the prefix equivalent of the following infix expression:

$$((17.4 - 8.9) * (3.9 - 4.7) + 18.4) / (3.8 - 7.9 * 2.3)$$

Chapter Summary

Recursion is an algorithmic technique in which a method calls itself. A method that uses recursion is called a recursive method.

Recursive methods include two cases: a base case that the method can solve directly without recursion, and a recursive case in which the method reduces a problem into a simpler problem of the same kind using a recursive call.

Recursive method calls work internally by storing information about each call into a structure called a call stack. When the method calls itself, information about the call is placed on top of the stack. When a method call finishes executing, its information is removed from the stack and the program returns to the call underneath.

A recursive method without a base case, or one in which the recursive case doesn't properly transition into the base case, can lead to infinite recursion.

A helper method is written to help solve a subtask of an overall problem. Recursive helper methods often have parameters in addition to the ones passed to the overall recursive method that calls them, to allow them to more easily implement the overall recursive solution.

Recursion can be used to draw graphical figures in complex patterns, including fractal images. Fractals are images that are recursively self-similar, and they are often referred to as "infinitely complex."

Self-Check Problems

Section 12.1: Thinking Recursively

1. What is recursion? How does a recursive method differ from a standard iterative method?
2. What are base cases and recursive cases? Why does a recursive method need to have both?
3. Consider the following method:

```
public static void mystery1(int n) {  
    if (n <= 1) {  
        System.out.print(n);  
    } else {  
        mystery1(n / 2);  
        System.out.print(", " + n);  
    }  
}
```

For each of the following calls, indicate the output that is produced by the method:

- a. `mystery1(1);`
- b. `mystery1(2);`
- c. `mystery1(3);`

- d. `mystery1(4);`
- e. `mystery1(16);`
- f. `mystery1(30);`
- g. `mystery1(100);`

4. Consider the following method:

```
public static void mystery2(int n) {  
    if (n > 100) {  
        System.out.print(n);  
    } else {  
        mystery2(2 * n);  
        System.out.print(", " + n);  
    }  
}
```

For each of the following calls, indicate the output that is produced by the method:

- a. `mystery2(113);`
- b. `mystery2(70);`
- c. `mystery2(42);`
- d. `mystery2(30);`
- e. `mystery2(10);`

5. Consider the following method:

```
public static void mystery3(int n) {  
    if (n <= 0) {  
        System.out.print("*");  
    } else if (n % 2 == 0) {  
        System.out.print("(");  
        mystery3(n - 1);  
        System.out.print(")");  
    } else {  
        System.out.print("[");  
        mystery3(n - 1);  
        System.out.print("]");  
    }  
}
```

For each of the following calls, indicate the output that is produced by the method:

- a. `mystery3(0);`
- b. `mystery3(1);`
- c. `mystery3(2);`
- d. `mystery3(4);`
- e. `mystery3(5);`

6. Consider the following method:

```
public void mysteryXY(int x, int y) {
    if (y == 1) {
        System.out.print(x);
    } else {
        System.out.print(x * y + ", ");
        mysteryXY(x, y - 1);
        System.out.print(", " + x * y);
    }
}
```

For each of the following calls, indicate the output that is produced by the method:

- a. `mysteryXY(4, 1);`
- b. `mysteryXY(4, 2);`
- c. `mysteryXY(8, 2);`
- d. `mysteryXY(4, 3);`
- e. `mysteryXY(3, 4);`

7. Convert the following iterative method into a recursive method:

```
// Prints each character of the string reversed twice.
// doubleReverse("hello") prints oolllleehh
public static void doubleReverse(String s) {
    for (int i = s.length() - 1; i >= 0; i--) {
        System.out.print(s.charAt(i));
        System.out.print(s.charAt(i));
    }
}
```

Section 12.2: A Better Example of Recursion

- 8. What is a call stack, and how does it relate to recursion?
- 9. What would be the effect if the code for the reverse method were changed to the following?

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        System.out.println(line);    // swapped order
        reverse(input);              // swapped order
    }
}
```

- 10. What would be the effect if the code for the reverse method were changed to the following?

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
```



```

        // recursive case (nonempty file)
        reverse(input);                // moved this line
        String line = input.nextLine();
        System.out.println(line);
    }
}

```

Section 12.3: Recursive Functions and Data

11. The following method is an attempt to write a recursive `pow` method to compute exponents. What is wrong with the code? How can it be fixed?

```

public static int pow(int x, int y) {
    return x * pow(x, y - 1);
}

```

12. What are the differences between the two versions of the `pow` method shown in Section 12.3? What advantage does the second version have over the first version? Are both versions recursive?

13. Consider the following method:

```

public static int mystery4(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return mystery4(x - y, y);
    }
}

```

For each of the following calls, indicate the value that is returned:

- a. `mystery4(6, 13)`
- b. `mystery4(14, 10)`
- c. `mystery4(37, 10)`
- d. `mystery4(8, 2)`
- e. `mystery4(50, 7)`

14. Consider the following method:

```

public static int mystery5(int x, int y) {
    if (x < 0) {
        return -mystery5(-x, y);
    } else if (y < 0) {
        return -mystery5(x, -y);
    } else if (x == 0 && y == 0) {
        return 0;
    } else {
        return 100 * mystery5(x / 10, y / 10) + 10 * (x % 10) + y % 10;
    }
}

```

For each of the following calls, indicate the value that is returned:

- a. `mystery5(5, 7)`
- b. `mystery5(12, 9)`
- c. `mystery5(-7, 4)`
- d. `mystery5(-23, -48)`
- e. `mystery5(128, 343)`

15. Consider the following method:

```
public static int mystery6(int n, int k) {  
    if (k == 0 || k == n) {  
        return 1;  
    } else if (k > n) {  
        return 0;  
    } else {  
        return mystery6(n - 1, k - 1) + mystery6(n - 1, k);  
    }  
}
```

For each of the following calls, indicate the value that is returned:

- a. `mystery6(7, 1)`
- b. `mystery6(4, 2)`
- c. `mystery6(4, 3)`
- d. `mystery6(5, 3)`
- e. `mystery6(5, 4)`

16. Convert the following iterative method into a recursive method:

```
// Returns n!, such as 5! = 1*2*3*4*5  
public static int factorial(int n) {  
    int product = 1;  
    for (int i = 1; i <= n; i++) {  
        product *= i;  
    }  
    return product;  
}
```

17. The following method has a bug that leads to infinite recursion. What correction fixes the code?

```
// Adds the digits of the given number.  
// Example: digitSum(3456) returns 3+4+5+6 = 18  
public static int digitSum(int n) {  
    if (n > 10) {  
        // base case (small number)  
        return n;  
    } else {
```



```

        // recursive case (large number)
        return n % 10 + digitSum(n / 10);
    }
}

```

18. Sometimes the parameters that a client would like to pass to a method don't match the parameters that are best for writing a recursive solution to the problem. What should a programmer do to resolve this issue?
19. The Fibonacci sequence is a sequence of numbers in which the first two numbers are 1 and each subsequent number is the sum of the previous two Fibonacci numbers. The sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on. The following is a correct, but inefficient, method to compute the n th Fibonacci number:

```

public static int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

```

The code shown runs very slowly for even relatively small values of n ; it can take minutes or hours to compute even the 40th or 50th Fibonacci number. The code is inefficient because it makes too many recursive calls. It ends up recomputing each Fibonacci number many times. Write a new version of this method that is still recursive and has the same header but is more efficient. Do this by creating a helper method that accepts additional parameters, such as previous Fibonacci numbers, that you can carry through and modify during each recursive call.

Section 12.4: Recursive Graphics

20. What is a fractal image? How does recursive programming help to draw fractals?
21. Write Java code to create and draw a regular hexagon (a type of polygon).

Section 12.5: Recursive Backtracking

22. Why is recursion an effective way to implement a backtracking algorithm?
23. What is a decision tree? How are decision trees important for backtracking?
24. Draw the decision tree that would have resulted for Figure 12.9 if the backtracking solution had explored NE first instead of last in the recursive explore method. (Hint: the tree changes at every level.)
25. The original North/East backtracking solution printed the following ways of traveling to (1, 2) in this order. In what order would they be printed if the solution had explored NE first instead of last?

```

moves: N N E
moves: N E N
moves: N NE
moves: E N N
moves: NE N

```

26. Figure 12.12 shows only part of the decision tree for the first two levels. How many entries are there at the second level of the full tree? How many are at level 3 of the full tree?

27. If our 8 Queens algorithm tried every possible square on the board for placing each queen, how many entries are there at the 8th and final level of the full tree? What does our algorithm do to avoid having to explore so many possibilities?
28. The 8 Queens `explore` method stops once it finds one solution to the problem. What part of the code causes the algorithm to stop once it finds a solution? How could the code be modified so that it would find and output every solution to the problem?

Exercises

1. Write a recursive method called `starString` that accepts an integer as a parameter and prints to the console a string of stars (asterisks) that is 2^n (i.e., 2 to the n^{th} power) long. For example,
- `starString(0)` should print `*` (because $2^0 == 1$)
 - `starString(1)` should print `**` (because $2^1 == 2$)
 - `starString(2)` should print `****` (because $2^2 == 4$)
 - `starString(3)` should print `*****` (because $2^3 == 8$)
 - `starString(4)` should print `*****` (because $2^4 == 16$)

The method should throw an `IllegalArgumentException` if passed a value less than 0.

2. Write a method called `writeNums` that takes an integer n as a parameter and prints to the console the first n integers starting with 1 in sequential order, separated by commas. For example, consider the following calls:

```
writeNums(5);
System.out.println(); // to complete the line of output
writeNums(12);
System.out.println(); // to complete the line of output
```

These calls should produce the following output:

```
1, 2, 3, 4, 5
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Your method should throw an `IllegalArgumentException` if passed a value less than 1.

3. Write a method called `writeSequence` that accepts an integer n as a parameter and prints to the console a symmetric sequence of n numbers composed of descending integers that ends in 1, followed by a sequence of ascending integers that begins with 1. The following table indicates the output that should be produced for various values of n :

Method call	Output produced
<code>writeSequence(1);</code>	1
<code>writeSequence(2);</code>	1 1
<code>writeSequence(3);</code>	2 1 2
<code>writeSequence(4);</code>	2 1 1 2
<code>writeSequence(5);</code>	3 2 1 2 3
<code>writeSequence(6);</code>	3 2 1 1 2 3
<code>writeSequence(7);</code>	4 3 2 1 2 3 4
<code>writeSequence(8);</code>	4 3 2 1 1 2 3 4
<code>writeSequence(9);</code>	5 4 3 2 1 2 3 4 5
<code>writeSequence(10);</code>	5 4 3 2 1 1 2 3 4 5

Notice that when n is odd the sequence has a single 1 in the middle, whereas for even values it has two 1s in the middle. Your method should throw an `IllegalArgumentException` if it is passed a value less than 1.

4. Write a recursive method called `doubleDigits` that accepts an integer n as a parameter and returns the integer obtained by replacing every digit of n with two of that digit. For example, `doubleDigits(348)` should return 334488. The call `doubleDigits(0)` should return 0. Calling `doubleDigits` on a negative number should return the negation of calling `doubleDigits` on the corresponding positive number; for example, `doubleDigits(-789)` should return -778899.
5. Write a recursive method called `writeBinary` that accepts an integer as a parameter and writes its binary representation to the console. For example, `writeBinary(44)` should print 101100.
6. Write a recursive method called `writeSquares` that accepts an integer parameter n and prints the first n squares separated by commas, with the odd squares in descending order followed by the even squares in ascending order. For example, `writeSquares(8);` prints the following output:

49, 25, 9, 1, 4, 16, 36, 64

A call of `writeSquares(1);` prints 1. The method should throw an `IllegalArgumentException` if it is passed a value less than 1.

7. Write a recursive method called `writeChars` that accepts an integer parameter n and that prints out a total of n characters. The middle character of the output should always be an asterisk (*). If you are asked to write out an even number of characters, then there will be two asterisks in the middle (***). Before the asterisk(s) you should write out less-than characters (<). After the asterisk(s) you should write out greater-than characters (>). Your method should throw an `IllegalArgumentException` if it is passed a value less than 1. For example, the following calls produce the following output:

Method call	Output produced
<code>writeChars(1);</code>	*
<code>writeChars(2);</code>	**
<code>writeChars(3);</code>	<*>
<code>writeChars(4);</code>	<***>
<code>writeChars(5);</code>	<<*>>
<code>writeChars(6);</code>	<<***>>
<code>writeChars(7);</code>	<<<*>>>
<code>writeChars(8);</code>	<<<***>>>

8. Write a recursive method called `multiplyEvens` that returns the product of the first n even integers. For example, `multiplyEvens(1)` returns 2 and `multiplyEvens(4)` returns 384 (because $2 * 4 * 6 * 8 = 384$). The method should throw an `IllegalArgumentException` if it is passed a value less than or equal to 0.
9. Write a recursive method called `sumTo` that accepts an integer parameter n and returns a real number representing the sum of the first n reciprocals. In other words, `sumTo( $n$ )` returns $(1 + 1/2 + 1/3 + 1/4 + \dots + 1/n)$. For example, `sumTo(2)` should return 1.5. The method should return 0.0 if it is passed the value 0 and throw an `IllegalArgumentException` if it is passed a value less than 0.

10. Write a recursive method called `digitMatch` that accepts two nonnegative integers as parameters and that returns the number of digits that match between them. Two digits match if they are equal and have the same position relative to the end of the number (i.e., starting with the ones digit). In other words, the method should compare the last digits of each number, the second-to-last digits of each number, the third-to-last digits of each number, and so forth, counting how many pairs match. For example, for the call of `digitMatch(1072503891, 62530841)`, the method would compare as follows, and return 4 because four of the pairs match (2-2, 5-5, 8-8, and 1-1).

```

1 0 7 2 5 0 3 8 9 1
  | | | | | | |
6 2 5 3 0 8 4 1

```

11. Write a recursive method called `repeat` that accepts a string `s` and an integer `n` as parameters and that returns `s` concatenated together `n` times. For example, `repeat("hello", 3)` returns "hellohellohello", and `repeat("ok", 1)` returns "ok", and `repeat("bye", 0)` returns "". String concatenation is an expensive operation, so for an added challenge try to solve this problem while performing fewer than `n` concatenations.
12. Write a recursive method called `isReverse` that accepts two strings as parameters and returns `true` if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalization), and `false` otherwise. For example, the call of `isReverse("hello", "eLLoH")` would return `true`. The empty string, as well as any one-letter string, is considered to be its own reverse.
13. Write a recursive method called `indexOf` that accepts two strings as parameters and that returns the starting index of the first occurrence of the second string inside the first string (or `-1` if not found). For example, the call of `indexOf("Barack Obama", "bam")` would return 8. (Strings already have an `indexOf` method, but you may not call it in your solution.)
14. Write a recursive method called `evenDigits` that accepts an integer parameter and that returns the integer formed by removing the odd digits from it. For example, `evenDigits(8342116)` returns 8426 and `evenDigits(-34512)` returns -42. If the number is 0 or has no even digits, such as 35159 or 7, return 0. Leading zeros in the result should be ignored.
15. Write a recursive method called `permut` that accepts two integers `n` and `r` as parameters and returns the number of unique permutations of `r` items from a group of `n` items. For given values of `n` and `r`, this value $P(n, r)$ can be computed as follows:

$$P(n, r) = \frac{n!}{(n - r)!}$$

For example, `permut(7, 4)` should return 840. It may be helpful to note that `permut(6, 3)` returns 120, or $840 / 7$.

16. The Sierpinski carpet is a fractal that is defined as follows: The construction of the Sierpinski carpet begins with a square. The square is cut into nine congruent subsquares in a 3-by-3 grid, with the central subsquare removed. The same process is then applied recursively to the eight other subsquares. Figure 12.14 shows the first few iterations of the carpet.

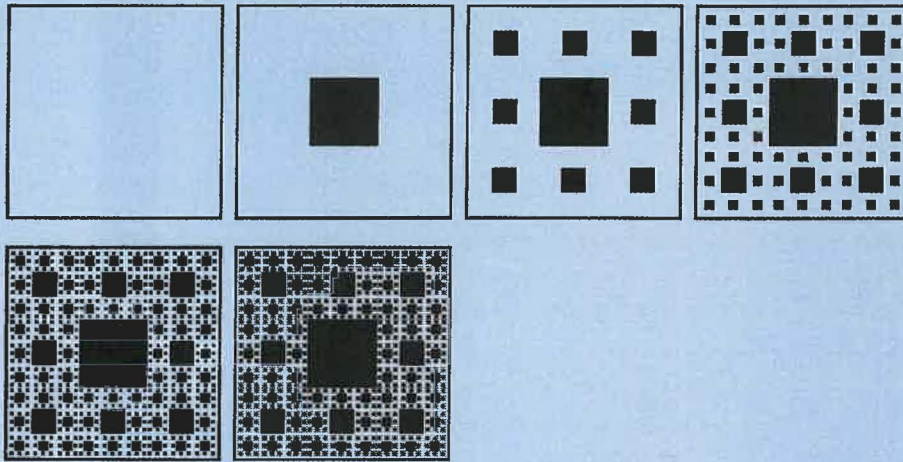


Figure 12.14 Sierpinski carpet

Write a program to draw the carpet on a `DrawingPanel` recursively.

17. The Cantor set is a fractal that is defined by repeatedly removing the middle thirds of line segments as shown in Figure 12.15.



Figure 12.15 Cantor set

Write a program to draw the Cantor set on a `DrawingPanel` recursively.

18. Write a recursive method called `waysToClimb` that takes a positive integer value representing a number of stairs and prints each unique way to climb a staircase of that height, taking strides of one or two stairs at a time. Do not use any loops. Output each way to climb the stairs on its own line, using a 1 to indicate a small stride of 1 stair, and a 2 to indicate a large stride of 2 stairs. The order in which you output the possible ways to climb the stairs is not important, so long as you list the right overall set of ways. For example, the call `waysToClimb(3);` should produce the following output:

```
[1, 1, 1]
[1, 2]
[2, 1]
```

The call `waysToClimb(4)`; should produce the following output:

```
[1, 1, 1, 1]
[1, 1, 2]
[1, 2, 1]
[2, 1, 1]
[2, 2]
```

19. Write a recursive method called `countBinary` that accepts an integer n as a parameter and that prints all binary numbers that have exactly n digits in ascending order, each on its own line. All n digits should be shown for all numbers, including leading zeros if necessary. Assume that n is nonnegative. If n is 0, a blank line should be produced.

Call	Output
<code>countBinary(1);</code>	0
	1
<code>countBinary(2);</code>	00
	01
	10
	11
<code>countBinary(3);</code>	000
	001
	010
	011
	100
	101
	110
	111

20. Write a recursive method called `subsets` to find every possible sub-list of a given list. A sub-list of a list L contains 0 or more of L 's elements. Your method should accept a list of strings as its parameter and print every sub-list that could be created from elements of that list, one per line. For example, if the list stores `{ Janet, Robert, Morgan, Char }`, the output from your method would be:

```
[Janet, Robert, Morgan, Char]
[Janet, Robert, Morgan]
[Janet, Robert, Char]
[Janet, Robert]
[Janet, Morgan, Char]
[Janet, Morgan]
[Janet, Char]
[Janet]
[Robert, Morgan, Char]
[Robert, Morgan]
```



```
[Robert, Char]
[Robert]
[Morgan, Char]
[Morgan]
[Char]
[]
```

The order in which you show the sub-lists does not matter, and the order of the elements of each sub-list also does not matter. The key thing is that your method should produce the correct overall set of sub-lists as its output. Notice that the empty list is considered one of these sub-lists. You may assume that the list passed to your method is not null and that the list contains no duplicates. Do not use any loops.

21. Write a recursive method called `maxSum` that accepts a list of integers, L , and an integer limit n as parameters and uses backtracking to find the maximum sum that can be generated by adding elements of L that do not exceed n . For example, if you are given the list $[7, 30, 8, 22, 6, 1, 14]$ and the limit of 19, the maximum sum that can be generated that does not exceed is 16, achieved by adding 7, 8, and 1. If the list L is empty, or if the limit is not a positive integer, or all of L 's values exceed the limit, return 0.

Each index's element in the list can be added to the sum only once, but the same number value might occur more than once in a list, in which case each occurrence might be added to the sum. For example, if the list is $[6, 2, 1]$ you may use up to one 6 in the sum, but if the list is $[6, 2, 6, 1]$ you may use up to two sixes.

List L	Limit n	Return Value
$[7, 30, 8, 22, 6, 1, 14]$	19	16
$[5, 30, 15, 13, 8]$	42	41
$[30, 15, 20]$	40	35
$[10, 20, 30]$	7	0
$[10, 20, 30]$	20	20
$[\]$	10	0

You may assume that all values in the list are nonnegative. Your method may alter the contents of the list L as it executes, but L should be restored to its original state before your method returns. Do not use any loops.

22. Write a recursive method called `printSquares` to find all ways to express an integer as a sum of squares of unique positive integers. For example, the call `printSquares(200);` should produce the following output:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
```

$$2^2 + 4^2 + 6^2 + 12^2$$

$$2^2 + 14^2$$

$$3^2 + 5^2 + 6^2 + 7^2 + 9^2$$

$$6^2 + 8^2 + 10^2$$

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your method should produce no output. Keep in mind that the sum has to be formed with unique integers. Otherwise you could always find a solution by adding 1^2 together until you got to whatever number you are working with.

As with any backtracking problem, this one amounts to a set of choices, one for each integer whose square might or might not be part of your sum. You may generate the choices by doing a `for` loop over an appropriate range of numbers. Note that the maximum possible integer that can be part of a sum of squares for an integer n is the square root of n .

Programming Projects

1. Write a recursive program to solve the “Missionaries and Cannibals” problem. Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How might they cross safely?

Your output should include the initial problem, the moves you make, and a “picture” of the current state of the puzzle after each move. Your final output should produce only the moves and the picture for the states that are on the solution path.

2. Write a recursive program to solve the Towers of Hanoi puzzle. The puzzle involves manipulating disks that you can move between three different towers. You are given a certain number of disks (four in this example) stacked on one of the three towers. The disks have decreasing diameters, with the smallest disk on the top (see Figure 12.16).

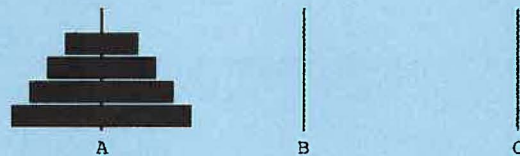


Figure 12.16 Towers of Hanoi

The object of the puzzle is to move all of the disks from one tower to another (say, from A to B). The third tower is provided as a temporary storage space as you move disks around. You are allowed to move only one disk at a time, and you are not allowed to place a disk on top of a smaller one (i.e., one with a smaller diameter).

Examine the rather simple solutions for one, two, and three disks, and see if you can discern a pattern. Then write a program that will solve the Towers of Hanoi puzzle for any number of disks. (*Hint:* Moving four disks is a lot like moving three disks, except that one additional disk is on the bottom.)

3. Write a recursive program to generate random sentences from a given BNF grammar. A BNF grammar is a recursively defined file that defines rules for creating sentences from tokens of text. Rules can be recursively self-similar. The following grammar can generate sentences such as "Fred honored the green wonderful child":

```

<s>::=<np> <vp>
<np>::=<dp> <adjp> <n>|<pn>
<dp>::=the|a
<adjp>::=<adj>|<adj> <adjp>
<adj>::=big|fat|green|wonderful|faulty|subliminal|pretentious
<n>::=dog|cat|man|university|father|mother|child|television
<pn>::=John|Jane|Sally|Spot|Fred|Elmo
<vp>::=<tv> <np>|<iv>
<tv>::=hit|honored|kissed|helped
<iv>::=died|collapsed|laughed|wept

```

4. The Koch snowflake is a fractal that is created by starting with a line segment, then recursively altering it as follows:

1. Divide the line segment into three segments of equal length.
2. Draw an equilateral triangle that has the middle segment from step 1 as its base.
3. Remove the line segment that is the base of the triangle from step 2.

Figure 12.17 shows the first several iterations of the snowflake.

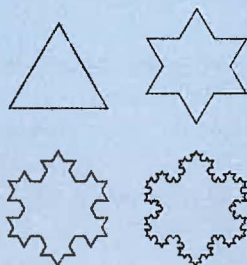


Figure 12.17 Koch snowflake

Write a program to draw the Koch snowflake on a `DrawingPanel` recursively.

5. Write a program that uses recursive backtracking to generate all anagrams from a phrase typed by the user. An anagram is a word or phrase made by rearranging the letters of another word or phrase. For example, the words "midterm" and "trimmed" are anagrams. If you ignore spaces and capitalization and allow multiple words, a multi-word phrase can be an anagram of some other word or phrase. For example, the phrases

“Clint Eastwood” and “old west action” are anagrams. Your program will read a dictionary file of words and search for all words that can be formed using the letters in the user’s phrase. Use backtracking to choose each word, explore what can be made out of the remaining letters, then un-choose the word afterward. Here is a possible example dialogue:

Phrase to search? **barbara bush**

Max words to use? **3**

[abash, bar, rub]

[abash, rub, bar]

[bar, abash, rub]

[bar, rub, abash]

[rub, abash, bar]

[rub, bar, abash]

Y	S	E	H
B	N	O	W
O	I	S	R
N	E	R	U

Figure 12.18 An example Boggle board

- Write a program that uses recursive backtracking to play the game of Boggle. Boggle is a word game played on a 4×4 grid where the player tries to find all valid dictionary words that can be made by tracing a path between adjacent letters from the board. Each link in the path can be horizontal, vertical, or diagonal. Figure 12.18 shows an example path to form the word “ensure”. Use recursive backtracking to explore each possible word that can be made using the letters on the board. Your algorithm should choose a starting square, explore what can be made from there, and un-choose the square afterward. (Hint: You will need a way to “mark” squares as being chosen or not chosen.)
- Write a program that uses recursive backtracking to find all ancestors and descendants of a person given a file of familial relationships. For ancestors it must show all parents, all grandparents, all great grandparents, etc. For descendants it must show all children, all grandchildren, all great grandchildren, etc. The program also must use indentation to make it clear who is a parent of whom and who is a child of whom. Write two recursive

methods that use backtracking to explore for ancestors and descendants respectively. Here is a possible example dialogue:

Input file? **tudor.dat**

Whose info? **Margaret**

Ancestors:

Margaret

Elizabeth of York

Henry VII

Descendants:

Margaret

James V

Mary, Queen of Scots

James VI & I

Margaret Stuart

Henry, Lord Darnley

James VI & I