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 substructure. 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:

```java
// This program prompts the user for a file or directory name
// and shows a listing of all files and directories that can be
// reached from it (including subdirectories).

import java.io.*;
import java.util.*;

public class DirectoryCrawler {
    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, 0);
        }
    }

    // Prints information for the given file/directory using the
    // given level of indentation
    public static void print(File f, int level) {
        for (int i = 0; i < level; i++) {
            System.out.print("    ");
        }
        System.out.println(f.getName());
        if (f.isDirectory()) {
            for (File subF : f.listFiles()) {
                print(subF, level + 1);
            }
        }
    }
}
```

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];
}
```