Class `File` 15

15.1 Class `File`

15.1.1 The File Path and the Instantiation of a `File` Object

As mentioned in Chap. 3, data can be read from text files with methods applied to `Scanner` objects instantiated with objects of type `File`. `File` is a class that represents files (including non-text files and file folders). To use `File` in a program, `File` must be imported by one of the following two import statements:

```
import java.io.File;
import java.io.*;
```

The $\star$ appearing the second import statement is a wildcard.

A `File` object can be instantiated with a file path as a parameter. A file path is a character sequence that represents a series of relative movements from a folder to another, starting with the present working folder and ending with a folder or a non-folder file (called a **regular file**).

Each "relative" movement specifies moving to a specific child folder, staying in the present folder, or moving to the parent folder. The specification of the child folder in the first case is by its name. The present folder and the parent folder are represented by `.` and `..`, respectively. For example, the folder name sequence

```
[ .., Documents, programs, test ]
```

specifies: moving to the parent folder, moving to the `Documents` folder, and then moving to the `programs` folder. The last entry `test` refers to a file or a folder named `test` residing in the folder `programs`. A `String` representation of a path is the concatenation of the movements and the name at the end with a special delimiter appearing in between. The delimiter is the backslash, `\`, in the case of the Windows and the forward slash, `/`, in the case of the others. Thus, the `String` representation of the file path is

```
"..\Documents\programs\test"
```

for the Windows, and

```
"../Documents/programs/test"
```

for the other systems. If the separator character appears consecutively (for instance, `///`), it is treated as just one separator.

The **absolute file path** to a file is the path that starts from the root of the file system. In the case of Windows, the root is specified by a single letter corresponding to the drive name (quite often, the drive name is `C`) followed by `:\`. In the case of other operating systems, the root is specified by the forward slash, `/`. An absolute file path may depend on how the file system is built and how the path is created in the program, so it may contain `.`, `..`, or consecutive appearances of the folder separator character. An absolute file path with such occurrences is redundant, and can be simplified into one that contains no such occurrences. We call such an absolute path without redundancy a **canonical path**. If `p` is a file path ending with a folder and `q` is a relative path, the concatenation `p` and `q` with the separator character between them is also a file path.

`File` has three constructors that receive a `String` data as a parameter. They are `File( String p )`, `File( File f, String p )`, and `File( String f, String p )`. In the last two, `f` and `p` are expected to represent a folder and a relative file path respectively. Let us take a look at an example. Suppose a file `records.txt` has the canonical path `/Users/maria/docs/records.txt` (in a non-Windows system), and this file is accessed from a Java program located in a folder whose canonical path is `/Users/maria/javacode`. Then, `File` objects representing `records.txt` can be instantiated using any one of the following `f0, ..., f5`:

```
File f0 = new File( "/Users/maria/docs/records.txt" );
File f1 = new File( "../docs/records.txt" );
File f2 = new File( new File( "../docs", "records.txt" ) );
File f3 = new File( "../docs", "records.txt" );
File f4 = new File( "../../maria", "docs/records.txt" );
File f5 = new File( "../../maria/docs", "records.txt" );
```

The absolute paths of the five `File` objects, in the order of appearance, are as follows:

```
f0: /Users/maria/docs/records.txt
f1: /Users/maria/javacode/../docs/records.txt
f2: /Users/maria/javacode/../docs/records.txt
f3: /Users/maria/javacode/../docs/records.txt
f4: /Users/maria/javacode/../../maria/docs/records.txt
f5: /Users/maria/javacode/../../maria/docs/records.txt
```

The instantiation of a `File` object does not guarantee the validity of the path, let alone the existence of the file specified by the path.

15.1.2 `File` Methods

Class `File` has many instance methods. Table 15.1 summarizes the instance methods that are covered in this chapter.

There are methods that inquire about the properties of the file represented by the `File` object. `exists` returns a `boolean` value representing whether or not the file indeed exists. `canRead` returns a `boolean` value representing if the file represented by the `File` object is readable. `canWrite` returns a `boolean` value representing if the file represented by the `File` object can be overwritten. `canExecute` returns a `boolean` value representing if the file represented by the `File` object can be executed. `isDirectory` returns a `boolean` value representing if the file represented by the `File` object is a folder. `isFile` returns a `boolean` value representing if the file represented by the `File` object is a regular file.

Table 15.1 A list of `File` methods

Name	Return type	Action
`exists()`	`boolean`	Returns whether or not `f` exists
`canRead()`	`boolean`	Returns whether or not `f` exists and can be read
`canWrite()`	`boolean`	Returns whether or not `f` exists and can be overwritten
`canExecute()`	`boolean`	Returns whether or not `f` exists and can be executed
`isDirectory()`	`boolean`	Returns whether or not `f` exists as a folder
`isFile()`	`boolean`	Returns whether or not `f` exists as a regular file
`getName()`	`String`	Returns the name of `f`
`getAbsolutePath()`	`String`	Returns the absolute path to `f`
`getAbsoluteFile()`	`File`	Returns a `File` object instantiated with the absolute path to `f`
`getCanonicalPath()`	`String`	Returns the canonical path to `f`
`getCanonicalFile()`	`File`	Returns a `File` object instantiated with the canonical path to `f`
`getParentPath()`	`String`	Return the paths to the parent of `f`, where the parent is determined based on the absolute path to `f`
`getParentFile()`	`File`	Returns the parent of `f` as a `File` object, where the parent is determined based on the absolute path to `f`
`length()`	`long`	Returns the byte size of `f`
`createNewFile()`	`boolean`	Attempts to create the file `f`; returns whether or not the attempt was successful
`delete()`	`boolean`	If `f` exists, attempts to remove `f`; returns whether or not the attempt was successful
`mkdir()`	`boolean`	Attempts to create `f` as a folder; returns whether or not the attempt was successful
`renameTo( File g )`	`boolean`	In the case where `f` exists, attempts to change the path to `f` to the path specified in `g`; returns whether or not the attempt was successful
`listFiles()`	`File[]`	If `f` exists and is a folder, return an array of `File` objects consisting of all the files in `f`; otherwise, return `null`

To simplify the description, we assume that the methods are applied to a `File` object `f`. All methods on this list take no parameters, except for `renameTo`

There is a group of `File` methods that provide information related to the file paths of the file represented by the `File` object. `getName` returns a `String` object that is equal to the name of the file. `getAbsolutePath` returns a `String` object representing the absolute path to the file represented by the `file` object. `getAbsoluteFile` returns a `File` object instantiated with the absolute path obtained by `getAbsolutePath`. `getCanonicalPath` returns a `String` object representing the canonical path to the file represented by the `file` object. `getCanonicalFile` returns a `File` object instantiated with the canonical path obtained by `getCanonicalPath`, `getParent` returns a `String` object representing the canonical path to the parent. `getParentFile` returns a `File` object instantiated with the path returned by `getParent`. Furthermore, if the `File` object represents a file folder, `listFiles` returns an array of `File` objects representing the files in the folder. If the `File` object does not represent a file folder, `listFiles` returns `null`.

Finally, there is a group of methods used for creating, removing, and renaming. `createNewFile` attempts to create the file represented by the `File` object as a regular file. `mkdir` attempts to create the file represented by the `File` object as a file folder. `delete()` attempts to remove the file represented by the `File` object. `renameTo( File g )` attempts to change the path to the file represented by the `File` object to that of `g`. These four methods return a `boolean` value representing whether or not the attempt was successful.

15.1.3 Exception Handling

15.1.3.1 Error Handling Imposed by the Compiler

A `throws` declaration is a declaration attached to the header of a method. It formally associates a run-time error type with the method. For an error type `E`, the declaration takes the form of `throws E` and appears immediately after the closing parenthesis of the parameter declaration in the method header. If a method `M` calls another method `N`, and `N` has a `throws E` declaration, then the Java compiler enforces that the code of `M` specifies how to handle a run-time error of type `E` originating from `N`. Thus, if a throws declaration in a method of an imported class, every method that calls the method must specify how to handle the error. One of the `File` methods introduced in Sect. 15.1.1, `createNewFile`, declares `throws IOException`.

Suppose we have the following program that attempts to create a folder named `"testDir"`, and then in the folder, attempts to create a file named `"testFile.txt"`.

```java
import java.io.*;
public class FileTest
{
  public static void main( String[] args )
  {
    File dir = new File( "testDir" );
    boolean success = dir.mkdir();
    File testFile = new File( dir, "testFile.txt" );
    success = testFile.createNewFile();
  }
}
```

An attempt to compile the program results in the following error message:

```
FileTest.java:9: error: unreported exception IOException; must be caught
    or declared to be thrown
    success = testFile.createNewFile();
                                    ^
1 error
```

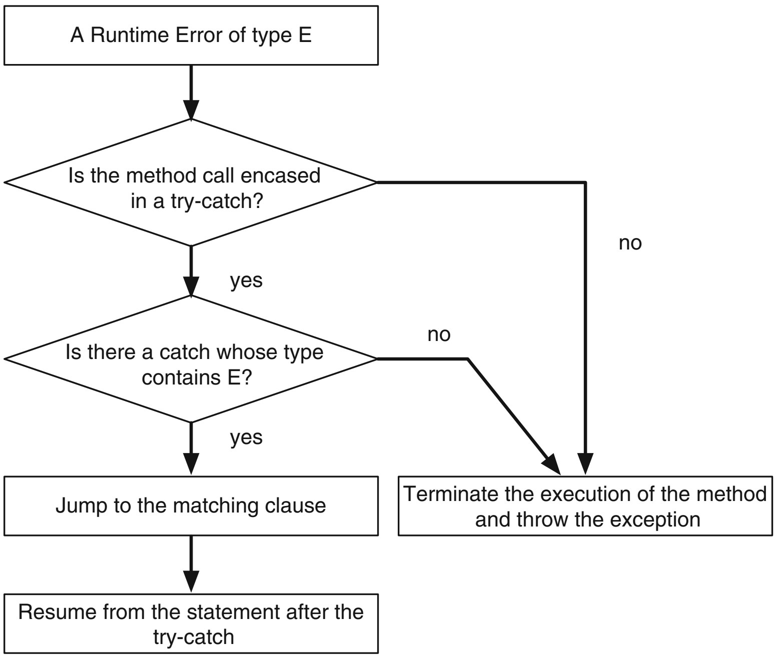
The message states that the method `createNewFile` has a `throws IOException` declaration but the present source code does not specify how to handle the error.

15.1.3.2 "Throwing" a Run-Time Error

There are two ways to handle a formally declared error that may result from a method call. One is to attach a `throws` declaration of the same error type to the method declaration:

```java
    public ... M( ... ) throws E
```

With this declaration, if an error of type `E` occurs during the execution of its code, `M` reports the error back to its direct superior. If the present execution of `M` is due to a method call to `M` appearing in another method `Q`, then the superior is `Q`. If there is no such `Q`, `M` must be the method `main` of some program, so the superior is JVM. In the latter case, JVM terminates the program after reporting the error on the screen.


A Runtime Error of type E
Is the method call encased in a try-catch?
no
yes
Is there a catch whose type contains E?
no
yes
Jump to the matching clause
Terminate the execution of the method and throw the exception
Resume from the statement after the try-catch
Fig. 15.1 The mechanism for handling run-time errors

A method cannot have more than one `throws` declaration. This limitation creates an issue when a method makes calls to multiple methods and they have different error types in their `throws` declarations. Fortunately, the Java run-time error types are organized as a tree, so any common ancestor of the different error types can be used as the error type in the `throws` declaration. `Exception` is the error type at the root of the tree, so in a situation where different run-time errors need to be handled, `Exception` can be used.

15.1.3.3 "Catching" a Run-Time Error

The other way to handle a formally declared run-time errors is to "catch" the error. The syntax for "catching" a run-time error of type `E` that originates from a method call to `N` is as follows: enclose a code block that includes the method call to `N` in a pair of curly brackets with a keyword `try` preceding the open bracket, and then immediately after the close bracket, add another pair of brackets preceded by a phrase `catch (E e)`. Some code can be placed inside the second pair of brackets. The lowercase letter `e` appearing in the parentheses represents an object representing the error that has occurred.

```
try
{
    ... N( ... )
}
catch (E e)
{
   ...
}
```

If an error of type `E` occurs during the execution of the `try` block, the execution of the block is immediately terminated, and then the execution of the `catch` block begins. After completing the code in the `catch` block, the execution jumps to the section immediately after the `catch` block. If an error of type `E` does not occur and no other run-time error occurs, the execution of the `try-`

block completes, and the execution reaches the section immediately after the catch block. Multiple catch clauses can be attached to one try block, if none of the error types appearing in the multiple catch clauses are ancestors of another.

Figure 15.1 summarizes the mechanism used to handle run-time errors.

Returning to the previous code example that failed to compile, any run-time error originating from createNewFile can be handled using a try-catch as follows:

```java
import java.io.*;
public class FileTestCatch
{
  public static void main( String[] args )
  {
    try
    {
      File dir = new File( "testDir" );
      boolean success = dir.mkdir();
      File testFile = new File( dir, "testFile.txt" );
      success = testFile.createNewFile();
    }
    catch ( IOException e )
    {
      System.out.println( "Error!" );
    }
  }
}
```

Listing 15.1 A program that demonstrates the use of try and catch

The difference between throwing and catching is that the former immediately terminates the execution of the method while the latter does not. Regardless of which approach is used to handle IOException, the class IOException needs to be imported. IOException is in the java.io package.

15.1.3.4 A Demonstration of Try-Catch

The following program demonstrates the use of try-catch for handling run-time errors.

The program receives two file paths from the user. The program instantiates a File object f with the first path and another File object g with the second path, and then executes a series of actions using two methods. The first method is called showExistence (Line 5). The method receives two File objects, f and g, as its formal parameters. The method prints information about the existence of the two files using the format "Existence: %s=%s, %s=%s%n" (Lines 7 and 8). The first and the third %s placeholders are for the file names, and the second and the fourth %s placeholders are for the return values of exists.

The second method of the program is called creationPlay (Line 10). This method has a throws IOException declaration. The method receives two File objects, f and g, as its formal parameters, and executes a series of actions. After each action, the method calls showExistence(f, g) to print the existence status of the two files. Here is the series of actions creationPlay performs:

1. Print the canonical paths of f and g obtained by calling getCanonicalPath (Lines 13 and 14).
2. Try to create the files specified by f and g using createNewFile (Lines 19 and 21).
3. Try to move f to itself using renameTo (Line 25).

```
 1  import java.util.*;
 2  import java.io.*;
 3  public class FileCreate
 4  {
 5    public static void showExistence( File f, File g )
 6    {
 7      System.out.printf( "Existence: %s=%s, %s=%s\n\n",
 8          f.getName(), f.exists(), g.getName(), g.exists() );
 9    }
```

Listing 15.2 A program that demonstrates the use of `File` methods for creating, deleting, and renaming files (part 1). The method `showExistence`

4. Try to move `f` to `g` using `renameTo` (Line 27).
5. Try to create the files specified by `f` and `g` as folders using `mkdir` (Lines 31–34).
6. Try to delete the files specified by `f` and `g` using `delete` (Lines 37–40).

In Steps 2 through 6, `creationPlay` announces the action it is about take. Since each method that is called returns a `boolean` value, the call is placed in `System.out.println`. This has the effect of printing the return value on the screen.

```
10    public static void creationPlay( File f, File g ) throws IOException
11    {
12      System.out.println( "The two input files" );
13      System.out.println( "File1 = " + f.getCanonicalPath() );
14      System.out.println( "File2 = " + g.getCanonicalPath() );
15      System.out.println( "----Their initial state" );
16      showExistence( f, g );
17
18      System.out.print( "----Create File1 -> " );
19      System.out.println( f.createNewFile() );
20      System.out.print( "----Create File2 -> " );
21      System.out.println( g.createNewFile() );
22      showExistence( f, g );
23
24      System.out.print( "----Rename File1 to File1 -> " );
25      System.out.println( f.renameTo( f ) );
26      System.out.print( "----Rename File1 to File2 -> " );
27      System.out.println( f.renameTo( g ) );
28      showExistence( f, g );
29
30      System.out.print( "----Create File1 as a folder -> " );
31      System.out.println( f.mkdir() );
32      System.out.print( "----Create File2 as a folder -> " );
33      System.out.println( g.mkdir() );
34      showExistence( f, g );
35
36      System.out.print( "----Delete File1 -> " );
37      System.out.println( f.delete() );
38      System.out.print( "----Delete File2 -> " );
39      System.out.println( g.delete() );
40      showExistence( f, g );
41    }
```

Listing 15.3 A program that demonstrates the use of `File` methods for creating, deleting, and renaming files (part 2). The method `creationPlay`

The method `main` (Line 42) receives two file paths from the user (Lines 46 and 47). The method instantiates a `File` object from each path, and then calls the method `creationPlay` with two `File` objects as actual parameters. Since the two `File` objects are not used anywhere else in the method `main`, this group of actions can be compressed into one line by placing the constructor calls inside the call to `creationPlay` (Line 50). A constructor does not have a return statement (see Chap. 16), but can be viewed as a method that returns an object of the type the constructor represents. The call appearing in Line 50 takes advantage of this fact. If the method call to `creationPlay` is successful, the method `main` prints a message (Line 51) and halts.

The program handles `IOException` that may originate from `creationPlay` (Line 53) using `try-catch`. If `creationPlay` produces a run-time error of `IOException`, the execution jumps to Line 53 without executing Line 51. There are two actions to be performed in the `catch` block. One is to print the information stored in `e`. This is accomplished by the method call `e.printStackTrace()` (Line 55). This method is available for all data types representing run-time errors. The method prints the series of method calls, starting from `main`, that have ultimately resulted in the run-time error at hand. The other action to be performed is to print a message to inform that a run-time error has occurred (Line 56). After producing the message on the screen, the program terminates because there is no statement after the `catch` block.

```
42    public static void main( String[] args )
43    {
44      Scanner keyboard = new Scanner( System.in );
45      System.out.print( "Enter two paths: " );
46      String p1 = keyboard.next();
47      String p2 = keyboard.next();
48      try
49      {
50        creationPlay( new File( p1 ), new File( p2 ) );
51        System.out.println( "Operations completed without errors" );
52      }
53      catch ( IOException e )
54      {
55        e.printStackTrace();
56        System.out.println( "---Program terminated---" );
57      }
58    }
59  }
```

Listing 15.4 A program that demonstrates the use of `File` methods for creating, deleting, and renaming files (part 3). The method `main`

Here are two examples of executing the program. The first example shows a case where the program executes the tasks without run-time errors.

```
 1  Enter two paths: tmp1 tmp2
 2  The two input files
 3  File1 = /Users/ogihara/file/tmp1
 4  File2 = /Users/ogihara/file/tmp2
 5  ----Their initial state
 6  Existence: tmp1=false, tmp2=false
 7
 8  ----Create File1 -> true
 9  ----Create File2 -> true
10  Existence: tmp1=true, tmp2=true
11
12  ----Rename File1 to File1 -> true
13  ----Rename File1 to File2 -> true
14  Existence: tmp1=false, tmp2=true
15
16  ----Create File1 as Directory -> true
17  ----Create File2 as Directory -> false
18  Existence: tmp1=true, tmp2=true
19
20  ----Delete File1 -> true
21  ----Delete File2 -> true
22  Existence: tmp1=false, tmp2=false
23
24  Operations completed without errors
```

Note that (i) immediately after moving the first file to the second file, the first file ceases to exist (Line 14), and (ii) the attempt to create the second file as a directory returns `false` (Line 17) because the second file already exists as a regular file.

The second example shows a case where the creation of the new file at the beginning of the series produces `IOException`. In this specific case, the error occurs because the directory `foo1` that appears in the first path does not exist.

```
 1  Enter two paths: foo1/foo2 foo3
 2  The two input files
 3  File1 = /Users/ogihara/file/foo1/foo2
 4  File2 = /Users/ogihara/file/foo3
 5  ----Their initial state
 6  Existence: foo2=false, foo3=false
 7  ----Create File1 -> java.io.IOException: No such file or directory
 8    at java.io.UnixFileSystem.createFileExclusively(Native Method)
 9    at java.io.File.createNewFile(File.java:1012)
10    at FileCreate.creationPlay(FileCreate.java:19)
11    at FileCreate.main(FileCreate.java:47)
12  ---Program terminated---
```

The printout generated by the call `e.printStackTrace()` starts in the middle of Line 7, after `"-> "` and ends in Line 11. Line 12 is the output produced by the print statement appearing in the `catch` block. Lines 8 through 10 present the chain of method calls that has led to the run-time error. The elements in the sequence appear one method call per line, starting from the most recent ones. By following the sequence from the end, we learn that the method `main` called the method `creationPlay` in Line 47 of the source code, the method `creationPlay` called the method `createNewFile` of `java.io.File` in Line 19 of the source code, the method `createNewFile` called the method `createFileExclusively` of `java.io.UnixFileSystem` in Line 1012 of the source code, and that method `createFileExclusively` generated the run-time error.

15.1.4 File Listing

The next program uses the method `listFiles` to take an inventory of the parent directory of the file specified in `args[ 0 ]`. After that, for each file appearing in the list, the program tests its properties using `canExecute`, `canRead`, `canWrite`, `isDirectory`, and `isFile`, and then reports the results. For each file, the program prints the results of the five tests. It also prints the name and the size of each file.

The first part of the source code is the method `directoryPlay`. The formal parameter of the method is a `File` object, `home` (Line 5). The method obtains the parent folder of `home` by using the method `getParentFile`, and stores the `File` object that is returned in a `File` variable named `parent` (Line 7). Next, the method obtains an array consisting of all the files residing in the folder `parent` using the method `listFiles`, and stores the returned array in an array of `File` objects named `list` (Line 8). The method then instantiates a rectangular array of `boolean` values named `flags`. In `flags`, the method stores the results of executing the five property tests for all files in the folder `parent`. The length of the first dimension of `flags` is the length of `list` and the length of the second dimension is 5, since the number of tests to be performed is 5 (Line 9). The method also instantiates an array of `long` values, `size` (Line 10), and an array of `String` values, `name` (Line 11). The method stores the file lengths in `size` and the file names in `name` for all files in the `list`. The length is thus `list.length` for both arrays.

After these array instantiations, the method examines the files appearing in the list. Using a for-loop, the method generates the index sequence `0, ..., list.length - 1` with a variable named `i`. At each round, the method executes the five tests, `canExecute`, `canRead`, `canRead`, `isDirectory`, and `isFile`, in this order, and stores the results in `flags[ i ][ 0 ], ..., flags[ i ][ 4 ]` (Lines 14–18). The program also obtains the file length and the file name using the methods `length` and `getName`, and stores the returned values in their respective arrays, `size` and `name`.

```
 1 import java.io.*;
 2 import java.util.*;
 3 public class FileExplore
 4 {
 5   public static void directoryPlay( File home )
 6   {
 7     File parent = home.getParentFile();
 8     File[] list = parent.listFiles();
 9     boolean[][] flags = new boolean[ list.length ][ 5 ];
10     long[] size = new long[ list.length ];
11     String[] name = new String[ list.length ];
12     for ( int i = 0; i < list.length; i ++ )
13     {
14       flags[ i ][ 0 ] = list[ i ].canExecute();
15       flags[ i ][ 1 ] = list[ i ].canRead();
16       flags[ i ][ 2 ] = list[ i ].canWrite();
17       flags[ i ][ 3 ] = list[ i ].isDirectory();
18       flags[ i ][ 4 ] = list[ i ].isFile();
19       size[ i ] = list[ i ].length();
20       name[ i ] = list[ i ].getName();
21     }
```

Listing 15.5 A program that demonstrates the use of `File` methods for examining properties of files (part 1). The part responsible for taking the inventory

The method prints the inventory thus obtained, but before that, the method prints the name of home, the absolute path to home, the name of parent, and the absolute path to parent that are obtained using the methods getName and getAbsolutePath (Lines 22–25). The method prints the inventory one file per line. The information that it prints consists of the index to the file in the file list, the five boolean values stored in the array flags, the file length, and the file name. The format it uses is the following:

```
%05d:%6s%6s%6s%6s%6s%8d %s%n
```

The first placeholder, %05d, is for the index, the next five occurrences of %6s are for the five boolean values, the next one, %8d, is for the file length, and the very last one, %s, is for the file name. To beautify the table, the program prints header lines (Line 26–28).

```
22      System.out.printf( "Home: %s%n", home.getName() );
23      System.out.printf( "Absolute Path: %s%n", home.getAbsolutePath() );
24      System.out.printf( "Parent: %s%n", parent.getName() );
25      System.out.printf( "Absolute Path: %s%n", parent.getAbsolutePath() );
26      System.out.println( "Parent's File List\n------------------" );
27      System.out.println(
28          "Index: Exec? Read?Write?IsDir?IsFil?    Size Name" );
29      for ( int i = 0; i < list.length; i ++ )
30      {
31        System.out.printf( "%05d:%6s%6s%6s%6s%6s%8d %s%n",
32            i, flags[ i ][ 0 ], flags[ i ][ 1 ], flags[ i ][ 2 ],
33            flags[ i ][ 3 ], flags[ i ][ 4 ], size[ i ], name[ i ] );
34      }
35    }
```

Listing 15.6 A program that demonstrates the use of File methods for examining properties of files (part 2). The part responsible for printing the results

The last part of the code is the method main. The method has a throws IOException declaration (Line 37). The program needs one file path to start. The actions that the method main performs are as follows:

- Ask the user to enter a path (Line 38).
- Instantiate a Scanner object (Line 39).
- Receive a file path from the user and store it in a String variable named path (Line 40).
- Instantiate a File object f with the path (Line 41).
- Instantiate a File object g with the canonical path obtained from f (Line 42).
- Call the method directoryPlay with g as the actual parameter.

```
36    public static void main( String[] args ) throws IOException
37    {
38      System.out.print( "Enter path: " );
39      Scanner keyboard = new Scanner( System.in );
40      String path = keyboard.nextLine();
41      File f = new File( path );
42      File g = f.getCanonicalFile();
43      directoryPlay( g );
44    }
45  }
```

Listing 15.7 A program that demonstrates the use of File methods for examining properties of files (part 3). The method main

Here is one execution example:

```
Enter path: ../..
Home: CSC120
Absolute Path: /Users/nancy/Documents/CSC120
Parent: Documents
Absolute Path: /Users/nancy/Documents
Parent's File List
------------------
Index: Exec? Read?Write?IsDir?IsFil?    Size Name
00000: false  true  true false  true   18436 .DS_Store
00001: false  true  true false  true       0 .localized
00002:  true  true  true  true false     476 archives
00003:  true  true  true  true false     204 classical.txt
00004:  true  true  true  true false     136 cranium
00005:  true  true  true  true false     714 CSC120
00006:  true  true  true  true false     578 CSC527
00007:  true  true  true  true false     204 easy
00008: false  true  true false  true  132102 eceSeminarOct2017.docx
00009:  true  true  true  true false     272 fiances
00010:  true  true  true  true false     306 frank
00011:  true  true  true  true false     612 globaltheme
00012:  true  true  true  true false     204 india.txt
00013:  true  true  true  true false     102 letters
00014:  true  true  true  true false     204 Microsoft User Data
00015:  true  true  true  true false     204 MitsuCollection.txt
00016: false  true  true false  true17438298 Mult_Pattern_2_EdXX.docx
00017:  true  true  true  true false     204 MyCollection.txt
00018:  true  true  true  true false     170 papers
00019:  true  true  true  true false     782 pdfs
00020:  true  true  true  true false     340 Projects
00021:  true  true  true  true false    1020 Resume
00022:  true  true  true  true false     442 reviews
00023:  true  true  true  true false     578 temp
00024:  true  true  true  true false     374 temporary
```

15.2 Using Scanner Objects to Read from Files

Suppose `theFile` is a `File` object that refers to a text file. Then, the `Scanner` object instantiated with `new Scanner( theFile )` makes it possible to read the contents of `theFile`. Since the constructors of `File` do not check the validity of the file paths provided, the file represented by `theFile` may not exist. In such a case, the constructor throws a run-time error `FileNotFoundException`. `FileNotFoundException` belongs to `java.io` and is a descendant of `IOException`. To handle `FileNotFoundException`, `java.io.FileNotFoundException` must be imported by one of the following two import statements:

```java
import java.io.*;
import java.io.FileNotFoundException;
```

The next program demonstrates how to read from a text file that contains scores for a group of students. The number of scores available differs from student to student. The file begins with an integer token that specifies the number of students whose scores appear in the file. This is followed by the information for individual students. The information for an individual student consists of the name of the student, the number of scores available for the students, and the individual scores for the student. The actual number of scores that appear must match the stated number of scores. Furthermore, the actual number of students whose records appear in the file must match the number of students stated in the first line of the file. The names are free of delimiters, so can be read with the method `next`.

Here is a sample score file:

```
8
Judy  5 80.5  82.0  85.0  92.0  95.0
King  6 89.0  87.5  77.5  100.0 95.5  98.0
Linda 3 94.5  95.5  96.5
Monty 4 75.5  80.0  90.0  65.0
Nancy 2 100.0 100.0
Oliver  1 99.0
Patty 3 97.5  95.0  92.5
Quincy  2 84.0  89.0
```

Listing 15.8 A sample data file for the score reading data program

Here is a program that reads a file whose contents are in this format. The program receives a path to the file from the user, and then stores it in a `String` variable named `path` (Lines 7–9). The program instantiates a `File` object with the file path, and then stores it in a variable named `theFile` (Line 10). The program instantiates a `Scanner` object `fileScanner` with `theFile` (Line 13). Since the instantiation of a `Scanner` object may result in `FileNotFoundException`, the instantiation is placed in a `try-catch`. The `try` block contains the instantiation of the `Scanner` object and all the remaining actions of the program. The corresponding `catch` appears much later (Line 37). The program reads the first token of the file using `nextInt`, and stores the number in an `int` variable named `lineNumber` (Line 14). The program instantiates an array of `String` data having length equal to `lineNumber`, and stores the array in a variable, `names` (Line 14). The program instantiates a jagged array having length `lineNumber`, and stores the array in a variable, `scores` (Line 15). Each row of `scores` is `null` immediately after the instantiation. The program then reads the data for individual students using a double for-loop. The exterior loop iterates over the indexes to the students, `0, ..., lineNumber - 1`, with a variable named `i` (Line 17). The internal loop iterates over the indexes to the scores (Line 22). The actions to be performed in the double for-loop are as follows:

- Read the name and store it in `names[ i ]` (Line 18).
- Read the number of scores, and store it in an `int` variable named `size` (Line 20).
- Instantiate the row `data[ i ]` with `new double[ size ]` (Line 21).
- Use a for-loop to read the individual scores and store them in `data[ i ][ 0 ], ..., data[ i ][ size - 1 ]` (Lines 22–25).

The number of students can be obtained with `names.length`. To obtain the number of scores available for the student at index `i`, `data[ i ].length` is used. To print the data, the program allocates ten character spaces for the name (Line 29), and ten character spaces with two digits after the decimal point (in other words, `%10.2f`) for each score (Line 32).

```java
import java.util.*;
import java.io.*;
public class ReadTable
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter file path: " );
    String path = keyboard.nextLine();
    File theFile = new File( path );
    try
    {
      Scanner fileScanner = new Scanner( theFile );
      int lineNumber = fileScanner.nextInt();
      String[] names = new String[ lineNumber ];
      double[][] data = new double[ lineNumber ][];
      for ( int i = 0; i < lineNumber; i ++ )
      {
        names[ i ] = fileScanner.next();
        int size = fileScanner.nextInt();
        data[ i ] = new double[ size ];
        for ( int col = 0; col < size; col ++ )
        {
          data[ i ][ col ] = fileScanner.nextDouble();
        }
      }
      for ( int i = 0; i < names.length; i ++ )
      {
        System.out.printf( "%10s", names[ i ] );
        for ( int col = 0; col < data[ i ].length; col ++ )
        {
          System.out.printf( "%10.2f", data[ i ][ col ] );
        }
        System.out.println();
      }
    }
    catch ( FileNotFoundException e )
    {
      System.out.printf( "The file %s does not exist%n", path );
    }
  }
}
```

Listing 15.9 A program that reads the names and scores of students from a file

Here is the result of executing the code on the data file presented earlier.

```
Enter file path: dataTable.txt
      Judy     80.50     82.00     85.00     92.00     95.00
      King     89.00     87.50     77.50    100.00     95.50     98.00
     Linda     94.50     95.50     96.50
     Monty     75.50     80.00     90.00     65.00
     Nancy    100.00    100.00
    Oliver     99.00
     Patty     97.50     95.00     92.50
    Quincy     84.00     89.00
```

The `try-catch` of the program is designed to handle `FileNotFoundException`. It does not handle other types of run-time errors. For example, consider a file `dataTableErroneous.txt` that was created by changing the number appearing at the very start of the file to 9 and retaining the remaining contents of the file. Since the number of students that is stated is larger than the actual number of students, the program results in a run-time error of `NoSuchElementException` as follows:

```
Enter file path: dataTableErroneous.txt
Exception in thread "main" java.util.NoSuchElementException
  at java.util.Scanner.throwFor(Scanner.java:862)
  at java.util.Scanner.next(Scanner.java:1371)
  at ReadTable.main(ReadTable.java:20)
```

However, if the user provides a file path to a non-existing file, the program prints the error message in the `catch` clause and halts as follows:

```
java ReadTable
Enter file path: fooBar.txt
The file fooBar.txt does not exist
% ...
```

People often use the "tab-delimited" format for data files. The following is a "tab-delimited" version of the data file we have just used, where the number of people or the numbers of scores does not appear. However, the information is presented one person per line and the entries in each line are separated by a tab-stop.

```
Judy   80.5  82.0  85.0  92.0  95.0
King   89.0  87.5  77.5  100.0 95.5  98.0
Linda  94.5  95.5  96.5
Monty  75.5  80.0  90.0  65.0
Nancy  100.0 100.0
Oliver  99.0
Patty  97.5  95.0  92.5
Quincy  84.0  89.0
```

Listing 15.10 A sample data file for the score reading data program. The file is without dimensional information

Suppose we must read from the tab-delimited version instead, and generate `names` and `data`, what should we do?

To decompose one line of data, we can choose from two strategies, as we discussed in Sect. 12.5. One strategy is to process the line twice with `Scanner`, the other is to use the method `split` of `String`. Regardless of which strategy is to be used, we must read the entire line using the method `nextLine`. Suppose we have just obtained one line of the file, and the student is at index `i`. Suppose that the line is stored in a `String` variable named `w`. With the first strategy, the performed actions will be as follows:

1. Instantiate a `Scanner` object with `w` as the parameter.
2. Run a while-loop with `hasNext` as the continuation condition, and obtain the number of tokens, say `m`, appearing in `w`.
3. Instantiate `data[ i ]` by `new double[ m - 1 ]`.
4. Instantiate the `Scanner` object again, and read the tokens from `w`. Store the very first token in `names[ i ]` and store the rest in `data[ i ]`.

The second strategy takes advantage of the fact that the entries in each line are separated by a tab. With that strategy, the actions to be performed are as follows:

1. Split `w` into an array of `String` data using `w.split( "\t" )`, and store it in an array, say `tokens`.
2. Store `tokens[ 0 ]` in `names[ i ]`.

3. Instantiate `data[ i ]` by `new double[ tokens.length - 1 ]`.
4. Using `Double.parseDouble`, convert `tokens[ 1 ], ..., tokens[ tokens.length - 1 ]` to double values, and then store them in `data[ i ][ 0 ], ..., data[ i ][ tokens.length - 2 ]`. `Double.parseDouble` is the `double` version of `Integer.parseInt`.

We use the latter strategy in our program.

The source code has the method `main` at the beginning. The method declares `throws IOException`. As before, the program obtains a file path from the user, and then instantiates a new `File` object (Lines 7–10). However, this time, the program verifies the existence and the readability of the specified file (Line 11). The program calls the method `theWork` only when the two tests return `true` (Line 13).

```
 1 import java.util.*;
 2 import java.io.*;
 3 public class ReadTableGuess
 4 {
 5   public static void main( String[] args ) throws FileNotFoundException
 6   {
 7     Scanner keyboard = new Scanner( System.in );
 8     System.out.print( "Enter file path: " );
 9     String path = keyboard.nextLine();
10     File theFile = new File( path );
11     if ( theFile.exists() && theFile.canRead() )
12     {
13       theWork( theFile );
14     }
15   }
16
```

Listing 15.11 A program that reads the score data from a file without dimensional information (part 1). The method `main`

The next method, `getLineNumber`, is for obtaining the number of lines in a file. The method `getLineNumber` executes the following series of action:

- Instantiate a `Scanner` object (Line 19).
- Store the value of 0 to an `int` variable `lineNumber` (Line 20).
- While there is a line remaining in the file (Line 21), read one line without saving it (Line 23) and increase `lineNumber` by 1 (Line 24).

```
17   public static int getLineNumber( File f ) throws FileNotFoundException
18   {
19     Scanner fileScanner = new Scanner( f );
20     int lineNumber = 0;
21     while ( fileScanner.hasNextLine() )
22     {
23       fileScanner.nextLine();
24       lineNumber ++;
25     }
26     fileScanner.close();
27     return lineNumber;
28   }
29
```

Listing 15.12 A program that reads the score data from a file without dimensional information (part 2). The method `getLineNumber`

- Close the Scanner object (Line 26) and return the value of lineNumber (Line 27).

The formal parameter of the method theWork is a File object named theFile. The method obtains the number of lines in the file with a call to getLineNumber, and then stores the returned value in its a variable, lineNumber (Line 32). The method instantiates the arrays names and data as before (Lines 33 and 34). The method then reads the individual lines. The line numbers are generated in the variable line. At each line, the method executes the following:

- Read one full line and store it in oneLine (Line 40).
- Call oneLine.split("\t") to obtain an array of String data, and store it in tokens (Line 41).
- Store tokens[0] in names[line] (Line 42).
- Store tokens.length - 1 in size (Line 43).
- Instantiate (data[line]) with new double[size] (Line 44).
- Use a for-loop that iterates over the sequence 0, ..., size - 1 with a variable named col, store Double.parseDouble(tokens[col + 1]) in data[line][col] (Lines 45–48).

After this, a section for printing the data appears. The source code is exactly the same as before.

```
30    public static void theWork( File theFile ) throws FileNotFoundException
31    {
32      int lineNumber = getLineNumber( theFile );
33      String[] names = new String[ lineNumber ];
34      double[][] data = new double[ lineNumber ][];
35
36      Scanner fileScanner = new Scanner( theFile );
37
38      for ( int line = 0; line < lineNumber; line ++ )
39      {
40        String oneLine = fileScanner.nextLine();
41        String[] tokens = oneLine.split( "\t" );
42        names[ line ] = tokens[ 0 ];
43        int size = tokens.length - 1;
44        data[ line ] = new double[ size ];
45        for ( int col = 0; col < size; col ++ )
46        {
47          data[ line ][ col ] = Double.parseDouble( tokens[ col + 1 ] );
48        }
49      }
50      for ( int row = 0; row < names.length; row ++ )
51      {
52        System.out.printf( "%10s", names[ row ] );
53        for ( int col = 0; col < data[ row ].length; col ++ )
54        {
55          System.out.printf( "%10.2f", data[ row ][ col ] );
56        }
57        System.out.println();
58      }
59    }
60  }
```

Listing 15.13 A program that reads the score data from a file without dimensional information (part 3). The part responsible for data reading and printing

15.3 Using PrintStream to Write to Files

Java has a number of classes for writing data to files. `PrintStream` is one of them. The class belongs to `java.io`, so the class must be imported. A `PrintStream` object can be instantiated with a `File` object as the parameter. Like the constructors of `Scanner`, the `PrintStream` constructor that takes a `File` object as the parameter has a `throws FileNotFoundException` declaration. This error occurs when the file path to the `File` object has a non-existing file folder. We are already very much familiar with `PrintStream`, because `System.out` has type `PrintStream`. We can use methods `print`, `printf`, and `println` on any `PrintStream` object that is instantiated with a `File` object. The way the methods work is exactly the same as the way they work for `System.out`. The only difference is that the output is generated not on the screen, but in a file.

Here is a program that converts the contents of a text file to all uppercase and saves the result in another file. The method `main` has a `throws FileNotFoundException` declaration (Line 6). The program obtains two file paths from the user (Lines 10 and 12), instantiates two `File` objects with the paths (Lines 14 and 15), opens the first file as an input file (Line 16), and opens the second file as an output file (Line 17). Then, the program enters a loop for converting the texts line by line (Line 18). The single statement appearing in the loop-body combines three actions: reading one line, applying `toUppercase` to the line, and writing the line in the output file (Line 20).

```
 1  import java.util.Scanner;
 2  import java.io.*;
 3  // Convert a file contents to all upper case
 4  public class ToUpper
 5  {
 6    public static void main( String[] args ) throws FileNotFoundException
 7    {
 8      Scanner keyboard = new Scanner( System.in );
 9      System.out.print( "Enter input file path: " );
10      String fInName = keyboard.next();
11      System.out.print( "Enter output file path: " );
12      String fOutName = keyboard.next();
13      assert !fInName.equals( fOutName ) : "The file names are equal!";
14      File fIn = new File( fInName );
15      File fOut = new File( fOutName );
16      Scanner scanner = new Scanner( fIn );
17      PrintStream stream = new PrintStream( fOut );
18      while ( scanner.hasNext() )
19      {
20        stream.println( scanner.nextLine().toUpperCase() );
21      }
22    }
23  }
```

Listing 15.14 A program that turns file contents of a file to all upper case and saves it to a new file

The Assert Statement In Line 13 of the program, we see

```
assert !fInName.equals( fOutName ): "The file names are equal!"
```

This line must be interpreted as:

if the condition `!fInName.equals( fOutName )` is `false` (equivalently, `fInName` is equal to `fOutName`), produce an error message `"The file names are equal!"`, and halt immediately after that.

This is almost identical to the `IllegalArgumentException` we have used before. The difference is that the `assert` statements are active only if the program is executed with a special option `-ea` of the command `java`. In other words, in the two executions

```
java -ea ToUpper
java ToUpper
```

the first one executes the `assert` statement and the second one ignores it. The type of a run-time error generated by `assert` is `AssertionError`. We show the result of executing the code on the following file. The file contains the lyrics to the original 1892 version of the song *America the Beautiful*:

```
O beautiful for halcyon skies,
For amber waves of grain,
For purple mountain majesties
Above the enameled plain!
America! America!
God shed His grace on thee,
Till souls wax fair as earth and air
And music-hearted sea!

O beautiful for pilgrim feet
Whose stern, impassioned stress
A thoroughfare for freedom beat
Across the wilderness!
America! America!
God shed His grace on thee
Till paths be wrought through wilds of thought
By pilgrim foot and knee!

O beautiful for glory-tale
Of liberating strife,
When once or twice, for man's avail,
Men lavished precious life!
America! America!
God shed His grace on thee
Till selfish gain no longer stain,
The banner of the free!

O beautiful for patriot dream
That sees beyond the years
Thine alabaster cities gleam
Undimmed by human tears!
America! America!
God shed His grace on thee
Till nobler men keep once again
Thy whiter jubilee!
```

Listing 15.15 The lyrics to *America the Beautiful*

Here is the result of executing the code:

```
O BEAUTIFUL FOR HALCYON SKIES,
FOR AMBER WAVES OF GRAIN,
FOR PURPLE MOUNTAIN MAJESTIES
ABOVE THE ENAMELED PLAIN!
AMERICA! AMERICA!
GOD SHED HIS GRACE ON THEE,
TILL SOULS WAX FAIR AS EARTH AND AIR
AND MUSIC-HEARTED SEA!

O BEAUTIFUL FOR PILGRIM FEET
WHOSE STERN, IMPASSIONED STRESS
A THOROUGHFARE FOR FREEDOM BEAT
ACROSS THE WILDERNESS!
AMERICA! AMERICA!
GOD SHED HIS GRACE ON THEE
TILL PATHS BE WROUGHT THROUGH WILDS OF THOUGHT
BY PILGRIM FOOT AND KNEE!

O BEAUTIFUL FOR GLORY-TALE
OF LIBERATING STRIFE,
WHEN ONCE OR TWICE, FOR MAN`''S AVAIL,
MEN LAVISHED PRECIOUS LIFE!
AMERICA! AMERICA!
GOD SHED HIS GRACE ON THEE
TILL SELFISH GAIN NO LONGER STAIN,
THE BANNER OF THE FREE!

O BEAUTIFUL FOR PATRIOT DREAM
THAT SEES BEYOND THE YEARS
THINE ALABASTER CITIES GLEAM
UNDIMMED BY HUMAN TEARS!
AMERICA! AMERICA!
GOD SHED HIS GRACE ON THEE
TILL NOBLER MEN KEEP ONCE AGAIN
THY WHITER JUBILEE!
```

When the program is run with the -ea option, if we provide identical file paths for the two files, the program halts with an error message as follows:

```
Enter input file path: foo
Enter output file path: foo
Exception in thread "main" java.lang.AssertionError: The file names are
    equal!
  at ToUpper.main(ToUpper.java:13)
```

When the program is run without the option of -ea, if we provide identical files paths for the files, the programs halts with an error, but the error message is different:

```
Enter input file path: foo
Enter output file path: foo
Exception in thread "main" java.io.FileNotFoundException: foo (No such
    file or directory)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:131)
  at java.util.Scanner.<init>(Scanner.java:611)
  at ToUpper.main(ToUpper.java:16)
```

Summary

- `File` is a class for specifying files with file paths. The class offers many methods for obtaining the properties of the files as well as obtaining paths to the files. They include `canRead`, `canWrite`, `isDirectory`, `isFile`, `length`, `listFiles`, `getName`, `getCanonicalPath`, and `getAbsolutePath`. Furthermore, the class offers methods for creating and removing actual files specified in `File` objects. They include `mkdir`, `delete`, and `createNewFile`.
- A `Scanner` object can be used to read data from a file specified by a `File` object.
- Instantiation of a `Scanner` object with a `File` object has a `throws FileNotFoundException` declaration.
- A `PrintStream` object can be used to write data to a file specified by a `File` object.
- `System.out` is a `PrintStream` object.
- The constructor of a `PrintStream` object with a `File` object as the parameter has a `throws FileNotFoundException` declaration.
- The declaration, `throws ERROR_TYPE`, attached to a method header officially announces that the method may throw a run-time error of the type. A method that calls such a method must handle a run-time error of the type generated as a result of the method call.
- Attachment of a `throws` declaration can be used to resolve the requirement to handle formally declared run-time errors.
- A `try-catch` clause can be used to handle run-time errors.
- Multiple `catch` clauses can be attached to a single `try` clause.
- An `assert` statement generates a run-time error if a condition does not hold.
- An `assert` is active only when the program that contains it is executed with the `-ea` option.
- The run-time errors in Java form a tree.

Exercises

1. **Questions about `File` methods**
   (a) Is the following statement true? If a `File` is instantiated with a file path and if the file specified the path does not exist, then `FileNotFoundException` is thrown.
   (b) Name the `File` method that returns whether or not a given file is a directory.
   (c) Name the `File` method that attempts to create a given file as a directory.
   (d) Name the `File` method that returns whether or not a given file exists.
   (e) Name the `File` method that returns whether or not a given file can be read.
   (f) Name the `File` method that returns whether or not a given file can be overwritten.
   (g) Name the `File` method that returns whether or not a given file can be executed.
   (h) Name the `File` method that, when the file specified by the `File` object is a folder, provides an array of `File` objects representing the files in the folder.
2. **Reading from a file** Suppose a file, `FooBar.txt`, consists of the following four lines:

```
a______10.5_
__bb_____20.5_
__
__ccc___-30.5
```

where each _ represents the whitespace. Suppose a `Scanner` object `reader` has been instantiated with a constructor call `new Scanner( new File( "FooBat.txt" ) )`, and then the following has been executed:

```
while ( reader.hasNext() )
{
  System.out.println(reader.next());
}
```

State how many lines of output are produced by this code.

3. **Reading a two-dimensional array** Write a program, `ReadMatrix`, that reads the elements of a two-dimensional matrix of floating point values stored in a file into a two-dimensional array, and then prints the elements of the array on the screen. The file name should be supplied in `args[ 0 ]`. Assume that, in the data file, the first token is the number of rows, the second token is the number of columns, and after these two tokens, the elements of the matrix appear in the row-major order. In the output, print the dimensions of the matrix, and then print the elements of the matrix, one row per line, with the entries appearing in the line same, where the entries appear with exactly three digits after the decimal point and exactly one whitespace in between. For example, if the contents of the input file are:

```
3
4
0.1
0.4
0.7
1.0
3.5
3.4
3.3
3.2
-1.0
1.0
-2.0
2.0
```

the output must look like:

```
3
4
0.100 0.400 0.700 1.000
3.500 3.400 3.300 3.200
-1.000 1.000 -2.000 2.000
```

4. **Reading a two-dimensional array and computing the row- and column-wise averages** Write a program, `MatrixAverage`, that reads the elements of a two-dimensional matrix of floating point values stored in a file into two-dimensional array, and then computes the row- and column-wise averages of the matrix. The file name is given as `args[ 0 ]`. The format used in the input file is:
   - the number of rows,
   - the number of columns, and
   - the entries of the matrix in the row-major order.
5. **Reading and printing** Write a program, `Punctuate`, that reads the tokens appearing in a text file, and prints the tokens on the screen as follows:
   - If the token ends with one of the following punctuation marks, print the newline after the token: the period, the exclamation mark, the question mark, the colon, the semicolon, and the comma.
   - Otherwise, print one whitespace after the token.

6. **Reading numbers while calculating their running total** Write a program, `RunningTotal`, that reads data from file, and computes and prints the running total of the numbers it encounters. All the tokens appearing in the file are integers. The program receives the file path from the user using `Scanner`.
7. **Searching for a key in a text file** Write a program named `ReadAndSearch` that receives a keyword and a file path from the user, and looks for all the occurrences of the key in the file. The program processes the contents of the file by reading them line by line. The program keeps track of the number of lines that have been read and uses it as the line number. Each time it finds that the line has an occurrence of the search key in it, the program prints the line number corresponding to the line.

Programming Projects

8. **Character counting** Write a program named `ASCIICounting` that receives a file path from the user, and then reads the contents of the file to produce the number of occurrences of each character in the region of characters with indexes 0–127 in the ASCII table. The `'\n'` appearing at the end of each line will be ignored when counting the character occurrences. After reading the file, the program prints the characters and their counts for all the characters that have been found in the file. The character with a count of 0 will be ignored in the output. The format to be used in producing the character-count pair on the screen is `'%c':%-6d`. The code must use a `try-catch` to handle the case in which the file path that the user specifies is not valid.

   Here is an execution example:

```
Enter file name: ASCIICounting.java
' ':402   '!':1     '"':6     '%':4     '&':2     ''':2
'(':20    ')':20    '*':2     '+':10    ',':2     '-':1
'.':17    '/':4     '0':9     '1':3     '2':3     '6':2
'7':2     '8':2     ':':1     ';':19    '<':3     '=':14
'>':2     'A':2     'C':3     'E':2     'F':4     'I':3
'L':1     'N':2     'O':1     'S':9     '[':10    ']':10
'a':29    'b':2     'c':19    'd':11    'e':32    'f':7
'g':8     'h':10    'i':58    'j':2     'l':12    'm':7
'n':46    'o':35    'p':20    'r':33    's':23    't':56
'u':13    'v':4     'w':7     'x':5     'y':6     '{':11
'}':11
```

9. **Character distribution** As an extension of the previous question, write a program `ASCIIDistComparison` that compares the ASCII character distributions of two files. The user specifies the paths to the files. For each file, the program produces a 128-dimensional array of `int` values representing the frequencies of the 128 characters. The program computes the difference between the two 128-dimensional arrays, and reports the differences on the screen. Use the same printing format as in the previous problem.

   Here is an execution example of such a program:

```
Enter file name 1: file1.txt
Enter file name 2: file2.txt
' ':7    '!':5    '"':-30  '%':5    '&':2    ''':8
'(':-10  ')':-10  '*':-6   '+':4    ',':-14  '-':-1
'.':-27  '/':9    '0':12   '1':5    '2':2    '3':-7
'4':-4   '5':-4   '6':2    '7':5    '8':4    '9':-7
':':-1   ';':-21  '<':2    '=':-6   '>':1    'A':3
'B':-8   'C':17   'D':5    'E':-1   'F':13   'G':-1
'I':-2   'L':-1   'M':-5   'N':9    'O':-3   'P':-2
'R':-10  'S':-19  'T':-3   'W':-4   '[':4    '\':-230
']':4    'a':-26  'b':-53  'c':2    'd':-18  'e':-63
```

```
'f':35   'g':-27  'h':-39  'i':8    'j':2    'k':-21
'l':-37  'm':-22  'n':-9   'o':13   'p':7    'q':-1
'r':-16  's':-31  't':-30  'u':-29  'v':-3   'w':-1
'x':3    'y':-27  '{':5    '|':-2   '}':5
```

10. **Character distribution plus** Revise the previous program to write a new one named `ASCIIDistComparisonPlus`. This time, the program additionally reports the character for which the difference is the largest.

    With the same input files, the output may be as follows:

```
Enter file name 1: file1.txt
Enter file name 2: file2.txt
' ':7    '!':5    '"':-30  '%':5    '&':2    ''':8
'(':-10  ')':-10  '*':-6   '+':4    ',':-14  '-':-1
'.':-27  '/':9    '0':12   '1':5    '2':2    '3':-7
'4':-4   '5':-4   '6':2    '7':5    '8':4    '9':-7
':':-1   ';':-21  '<':2    '=':-6   '>':1    'A':3
'B':-8   'C':17   'D':5    'E':-1   'F':13   'G':-1
'I':-2   'L':-1   'M':-5   'N':9    'O':-3   'P':-2
'R':-10  'S':-19  'T':-3   'W':-4   '[':4    '\':-230
']':4    'a':-26  'b':-53  'c':2    'd':-18  'e':-63
'f':35   'g':-27  'h':-39  'i':8    'j':2    'k':-21
'l':-37  'm':-22  'n':-9   'o':13   'p':7    'q':-1
'r':-16  's':-31  't':-30  'u':-29  'v':-3   'w':-1
'x':3    'y':-27  '{':5    '|':-2   '}':5
Max. difference of -230 for '\'.
```

11. **Printing the text shape to a file** Write a program, `TextShape`, that reads a text file and prints the "shape" of the text line into another file. For a `String` data `w`, by "shape" of `w`, we mean a `String` data generated from `w` by converting each character occurring between the first non-space character and the last non-space character to `'#'`.

    The user specifies the input file and the output file to execute this program.

    For example, the "shape" of

```
A BCDEFG
 HIJKLM
  NOPQ RS T UV
W X Y Z
```

is

```
#######
 ######
  ############
#######
```

Make sure that the program works when the input file contains a nonempty line consisting solely of `' '`.

12. **Converting each tab-stop to an equivalent series of `' '`** Write a program named `TabConvert` that reads from a file and produces an output where each tab-stops appearing in the input file is substituted with a series of `' '`, so that when printed the appearance of the input file and the appearance of the output are exactly the same. Assume that the tab-stop positions are multiples of 8. The program receives the paths to the input and output files from the user. In the program, write a method `convert` whose action is to receive a `String` value without the newline character as the formal parameter, and then return a tab-stop free `String` value that has the same appearance as the parameter when printed with no indentation.