Chapter 12

Text I/O

Objectives

- To discover file/directory properties, to delete and rename files/directories, and to create directories using the **File** class
- To write data to a file using the **PrintWriter** class
- To read data from a file using the **Scanner** class

The **File** Class: an introduction

- *The* **File** *class contains the methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory.*
- To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device.
- The file can then be transported and read later by other programs. Since data are stored in files, this section introduces how to use the **File** class to obtain file/directory properties, to delete and rename files/directories, and to create directories.

Absolute File Name vs Relative File Name

- Every file is placed in a directory in the file system. An *absolute file name* (or *full name*) contains a file name with its complete path and drive letter.
- For example, **c:\book\ Welcome.java** is the absolute file name for the file **Welcome.java** on the Windows operating system. Here **c:\book** is referred to as the *directory path* for the file.
- Absolute file names are machine dependent. On the UNIX platform, the absolute file name may be **/home/liang/book/Welcome.java**, where **/home/liang/book** is the directory path for the file **Welcome.java**.
- A *relative file name* is in relation to the current working directory. The complete directory path for a relative file name is omitted. For example, **Welcome.java** is a relative file name. If the current working directory is **c:\book**, the absolute file name would be **c:\book\Welcome.java**.

The File Class: Creates a File object for the specified path name

- **new File(String pathname)** creates a new File instance by converting the given pathname string into an abstract pathname.
- **new File("c:\\book")** creates a **File** object for the directory **c:\book**, and **new File("c:\\book\\test.dat")** creates a **File** object for the file **c:\book\test.dat**, both on Windows.
- Note that the directory separator for Windows is a backslash (\). The backslash is a special character in Java and should be written as **** in a string literal.
- Do not use absolute file names in your program. If you use a file name such as **c:\\book\\Welcome.java**, it will work on Windows but not on other platforms.
- You should use a file name relative to the current directory. For example, you may create a **File** object using **new File("Welcome.java")** for the file **Welcome.java** in the current directory. You may create a **File** object using **new File("image/us.gif")** for the file **us.gif** under the **image** directory in the current directory.
- The forward slash (/) is the Java directory separator, which is the same as on UNIX. The statement **new File("image/us.gif")** works on Windows, UNIX, and any other platform.

Methods in the File class

- Listing 12.12 demonstrates how to create a **File** object and use the methods in the **File** class to obtain its properties.
- The program creates a **File** object for the file **us.gif**. This file is stored under the **image** directory in the current directory.
- The **lastModified()** method returns the date and time when the file was last modified, measured in milliseconds since the beginning of UNIX time (00:00:00 GMT, January 1, 1970). The **Date** class is used to display it in a readable format in lines 14–15.
- Note that constructing a **File** instance does not create a file on the machine.
- You can create a **File** instance for any file name regardless whether it exists or not.
- You can invoke the **exists()** method on a **File** instance to check whether the file exists.
- So, executing TestFileClass does not create the file on the machine yet.

LISTING 12.12 TestFileClass.java

```
1  public class TestFileClass {
2    public static void main(String[] args) {
3      java.io.File file = new java.io.File("image/us.gif");
4      System.out.println("Does it exist? " + file.exists());
5      System.out.println("The file has " + file.length() + " bytes");
6      System.out.println("Can it be read? " + file.canRead());
7      System.out.println("Can it be written? " + file.canWrite());
8      System.out.println("Is it a directory? " + file.isDirectory());
9      System.out.println("Is it a file? " + file.isFile());
10     System.out.println("Is it absolute? " + file.isAbsolute());
11     System.out.println("Is it hidden? " + file.isHidden());
12     System.out.println("Absolute path is " +
13       file.getAbsolutePath());
14     System.out.println("Last modified on " +
15       new java.util.Date(file.lastModified()));
16   }
17 }
```

```
C:\book>java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is C:\book\image\us.gif
Last modified on Tue Nov 02 08:20:45 EST 2004

C:\book>_
```

File.createNewFile()

- The **createNewFile()** function is a part of File class in Java .
- This function creates a new empty file.
- The function returns true if the abstract file path does not exist and a new file is created.
- It returns false if the filename already exists.
- If File.exists() return true, it means that the file exists and createNewFile() will always return false until the file is deleted.

createNewFile() may throw an I/O exception. Java forces you to write the code to deal with this type of exception

```
1  import java.io.File;
2  import java.io.IOException;
3
4  public class TestFileClass {
5
6      public static void main(String[] args) throws IOException {
7          // TODO Auto-generated method stub
8          //java.io.File file = new java.io.File("image/us.gif");
9
10         java.io.File file = new File("Welcome.txt");
11
12         boolean fileExist = file.exists();
13
14         if (fileExist) {
15             System.out.println("(1) The file has " + file.length() + " bytes");
16             System.out.println("(2) Can it be read? " + file.canRead());
17             System.out.println("(3) Can it be written? " + file.canWrite());
18             System.out.println("(4) Is it a directory? " + file.isDirectory());
19             System.out.println("(5) Is it a file? " + file.isFile());
20             System.out.println("(6) Is it absolute? " + file.isAbsolute());
21             System.out.println("(7) Is it hidden? " + file.isHidden());
22             System.out.println("(8) Absolute path is " +
23                     file.getAbsolutePath());
24         } else {
25             System.out.println("Last modified on " +
26                     new java.util.Date(file.lastModified()));
27         }
28
29         System.out.println("Created successfully? " + file.createNewFile());
30
31     }
32
33 }
```


File Input and Output

- *Use the* **Scanner** *class for reading text data from a file and the* **PrintWriter** *class for writing text data to a file. (*Binary files are discussed in Chapter 17.)
- The **java.io.PrintWriter** class can be used to create a file and write data to a text file.
- First, you have to create a **PrintWriter** object for a text file as follows:
  PrintWriter output = **new** PrintWriter(filename);
- Then, you can invoke the **print**, **println**, and **printf** methods on the **PrintWriter** object to write data to a file.

java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter object for the specified file object.
+PrintWriter(filename: String)	Creates a PrintWriter object for the specified file-name string.
+print(s: String): void	Writes a string to the file.
+print(c: char): void	Writes a character to the file.
+print(cArray: char[]): void	Writes an array of characters to the file.
+print(i: int): void	Writes an int value to the file.
+print(l: long): void	Writes a long value to the file.
+print(f: float): void	Writes a float value to the file.
+print(d: double): void	Writes a double value to the file.
+print(b: boolean): void	Writes a boolean value to the file.
Also contains the overloaded println methods.	A println method acts like a print method; additionally, it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output."

FIGURE 12.8 The PrintWriter class contains the methods for writing data to a text file.

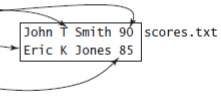
Writing Data Using **PrintWriter**

- Listing 12.13 gives an example that creates an instance of **PrintWriter** and writes two lines to the file **scores.txt**.
- Invoking the constructor of **PrintWriter** will create a new file if the file does not exist.
- If the file already exists, the current content in the file will be discarded without verifying with the user.
- In the code, line 6 uses System.exit(1) to terminate the program if file already exists. In this way, it will NOT overwrite the content of the existing file.
- Invoking the constructor of **PrintWriter** may throw an I/O exception. Java forces you to write the code to deal with this type of exception.
- For simplicity, we declare **throws IOException** in the main method header (line 2).
- The **close()** method must be used to close the file (line 19). If this method is not invoked, the data may not be saved properly in the file.

To simply use new File("scores.txt"), we have to add the line import java.io.File;

LISTING 12.13 WriteData.java

```
1 public class WriteData {
2   public static void main(String[] args) throws IOException {
3     java.io.File file = new java.io.File("scores.txt");
4     if (file.exists()) {
5       System.out.println("File already exists");
6       System.exit(1);
7     }
8
9     // Create a file
10    java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12    // Write formatted output to the file
13    output.print("John T Smith ");
14    output.println(90);
15    output.print("Eric K Jones ");
16    output.println(85);
17
18    // Close the file
19    output.close();
20  }
21 }
```



You have used the **System.out.print**, **System.out.println**, and **System.out.printf** methods to write text to the console. **System.out** is a standard Java object for the console output.
You can create **PrintWriter** objects for writing text to any file using **print**, **println**, and **printf** (lines 13–16).

Reading Data Using **Scanner**

- The **java.util.Scanner** class was used to read strings and primitive values from the console in Section 2.3, Reading Input from the Console.
- A **Scanner** breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a **Scanner** for **System.in**, as follows:
  Scanner input = **new** Scanner(System.in);
- To read from a file, create a **Scanner** for a file, as follows:
  Scanner input = **new** Scanner(**new** File(filename));

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that scans tokens from the specified file.
+Scanner(source: String)	Creates a Scanner that scans tokens from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

FIGURE 12.9 The Scanner class contains the methods for scanning data.

Reading Data Using **Scanner: an example**

- Listing 12.15 gives an example that creates an instance of **Scanner** and reads data from the file **scores.txt**.
- Invoking the constructor **new Scanner(File)** may throw an I/O exception, so the **main** method declares **throws Exception** in line 4. One possibility is attempting to create a Scanner object for a nonexistent file.
- It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file.
- The **nextByte()**, **nextShort()**, **nextInt()**, **nextLong()**, **nextFloat()**, **nextDouble()**, and **next()** methods are known as *token-reading methods*, because they read tokens separated by delimiters (by default, whitespace characters).
- For the **next()** method, no conversion is performed. If the token does not match the expected type, a runtime exception **java.util.InputMismatchException** will be thrown.

LISTING 12.15 ReadData.java

```
1  import java.util.Scanner;
2
3  public class ReadData {
4    public static void main(String[] args) throws Exception {
5      // Create a File instance
6      java.io.File file = new java.io.File("scores.txt");
7
8      // Create a Scanner for the file
9      Scanner input = new Scanner(file);
10
11     // Read data from a file
12     while (input.hasNext()) {
13       String firstName = input.next();
14       String mi = input.next();
15       String lastName = input.next();
16       int score = input.nextInt();
17       System.out.println(
18         firstName + " " + mi + " " + lastName + " " + score);
19     }
20
21     // Close the file
22     input.close();
23   }
24 }
```

scores.txt

John T Smith 90
Eric K Jones 85

Try running this program without the existence of "scores.txt" to see what happens.

next() and nextLine()

- Both methods **next()** and **nextLine()** read a string. The **next()** method reads a string delimited by delimiters, and **nextLine()** reads a line ending with a line separator.
- The token-reading method does not read the delimiter after the token.
- If the **nextLine()** method is invoked after a token-reading method, this method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of thestring returned by **nextLine()**.
- Suppose a text file named **test.txt** contains a line **34 567**
- After the following code is executed,

```
Scanner input = new Scanner(new File("test.txt"));
int intValue = input.nextInt();
String line = input.nextLine();
```

  - **intValue** contains **34** and **line** contains the characters **' '**, **5**, **6**, and **7**.

What happens if the input is *entered from the keyboard*? Suppose you enter **34**, press the *Enter* key, then enter **567** and press the *Enter* key for the following code:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
String line = input.nextLine();
```

Case Study: Replacing Text

- Suppose you are to write a program named **ReplaceText** that replaces all occurrences of a string in a text file with a new string.
- The file name and strings are passed as command-line arguments as follows: **java ReplaceText sourceFile targetFile oldString newString**
- For example, invoking

  java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
  - replaces all the occurrences of **StringBuilder** by **StringBuffer** in the file **FormatString .java** and saves the new file in **t.txt**.

Case Study: Replacing Text

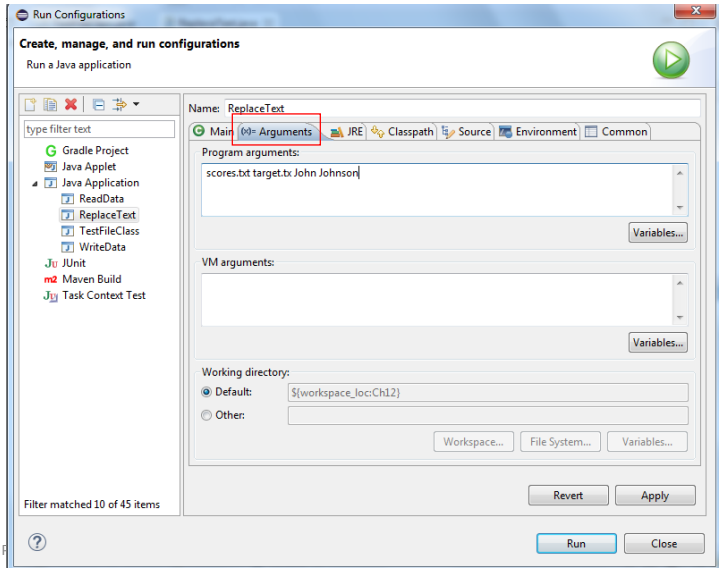
- Listing 12.16 gives the program. The program performs the following actions:
  - checks the number of arguments passed to the **main** method (lines 7–11),
  - checks whether the source and target files exist (lines 14–25),
  - creates a **Scanner** for the source file (line 29),
  - creates a **PrintWriter** for the target file (line 30), and
  - repeatedly reads a line from the source file (line 33), replaces the text (line 34), and writes a new line to the target file (line 35).
- The exit status code 1, 2, and 3 are used to indicate the abnormal terminations (lines 10, 17, 24).

LISTING 12.16 ReplaceText.java

```
1  import java.io.*;
2  import java.util.*;
3
4  public class ReplaceText {
5    public static void main(String[] args) throws Exception {
6      // Check command line parameter usage
7      if (args.length != 4) {
8        System.out.println(
9          "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10       System.exit(1);
11     }
12
13     // Check if source file exists
14     File sourceFile = new File(args[0]);
15     if (!sourceFile.exists()) {
16       System.out.println("Source file " + args[0] + " does not exist");
17       System.exit(2);
18     }
19
20     // Check if target file exists
21     File targetFile = new File(args[1]);
22     if (targetFile.exists()) {
23       System.out.println("Target file " + args[1] + " already exists");
24       System.exit(3);
25     }
26
27     try (
28       // Create input and output files
29       Scanner input = new Scanner(sourceFile);
30       PrintWriter output = new PrintWriter(targetFile);
31     ) {
32       while (input.hasNext()) {
33         String s1 = input.nextLine();
34         String s2 = s1.replaceAll(args[2], args[3]);
35         output.println(s2);
36       }
37     }
38   }
39 }
```

Running Listing 12.16 in Eclipse

- Open Run → *Run Configurations..* window.
- At the configuration tabs, go to *Arguments* tab.
- There you can specify the arguments to the program.



Chapter Summary

- The **File** class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.
- You can use **Scanner** to read string and primitive data values from a text file and use **PrintWriter** to create a file and write data to a text file.