12.11 File Input and Output

Key Point

Use the **`Scanner`** *class for reading text data from a file, and the* **`PrintWriter`** *class for writing text data to a file.*

VideoNote
Write and read data

A `File` object encapsulates the properties of a file or a path, but it does not contain the methods for writing/reading data to/from a file (referred to as data *input* and *output*, or *I/O* for short). In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. There are two types of files: text and binary. Text files are essentially characters on disk. This section introduces how to read/write strings and numeric values from/to a text file using the `Scanner` and `PrintWriter` classes. Binary files will be introduced in Chapter 17.

12.11.1 Writing Data Using `PrintWriter`

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. Figure 12.8 summarizes frequently used methods in `PrintWriter`.

`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.

Listing 12.13 gives an example that creates an instance of `PrintWriter` and writes two lines to the file **scores.txt**. Each line consists of a first name (a string), a middle-name initial (a character), a last name (a string), and a score (an integer).

LISTING 12.13 `WriteData.java`

throws an exception
create `File` object
file exist?

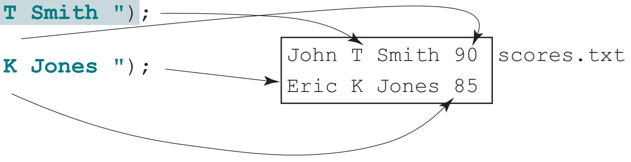
```
public class WriteData {
  public static void main(String[] args) throws java.io.IOException {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(1);
    }

```

```
 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 }
```

create `PrintWriter`

print data



```
John T Smith 90
Eric K Jones 85
```

close file

Lines 4–7 check whether the file **scores.txt** exists. If so, exit the program (line 6).

create a file

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.

`throws IOException`

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).

`print` method

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

close file

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.

Note
You can append data to an existing file using new PrintWriter(new FileOutputStream(file, true)) to create a PrintWriter object. FileOutputStream will be introduced in Chapter 17.

Tip
When the program writes data to a file, it first stores the data temporarily in a buffer in the memory. When the buffer is full, the data are automatically saved to the file on the disk. Once you close the file, all the data left in the buffer are saved to the file on the disk. Therefore, you must close the file to ensure that all data are saved to the file.

12.11.2 Closing Resources Automatically Using try-with-resources

Programmers often forget to close the file. JDK 7 provides the following try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {
  Use the resource to process the file;
}
```

Using the try-with-resources syntax, we rewrite the code in Listing 12.13 as shown in Listing 12.14.

Listing 12.14 WriteDataWithAutoClose.java

```
public class WriteDataWithAutoClose {
  public static void main(String[] args) throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
```

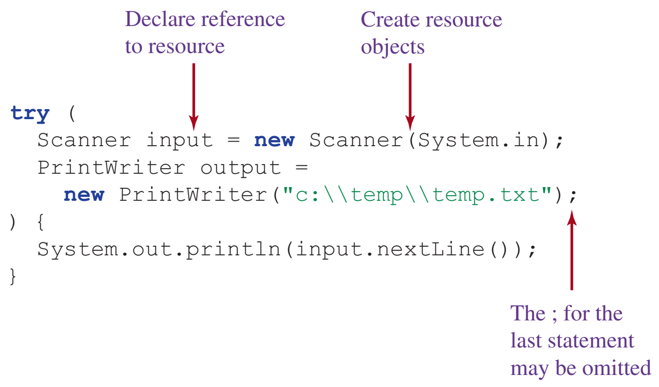
declare/create resource

use the resource

```
 7       }
 8
 9       try (
10         // Create a file
11         java.io.PrintWriter output = new java.io.PrintWriter(file);
12       ) {
13         // Write formatted output to the file
14         output.print("John T Smith ");
15         output.println(90);
16         output.print("Eric K Jones ");
17         output.println(85);
18       }
19     }
20   }
```

A resource is declared and created in the parentheses following the keyword `try`. The resources must be a subtype of `AutoCloseable` such as a `PrinterWriter` that has the `close()` method. A resource must be declared and created in the same statement, and multiple resources can be declared and created inside the parentheses. The statements in the block (lines 12–18) immediately following the resource declaration use the resource. After the block is finished, the resource's `close()` method is automatically invoked to close the resource. Using try-with-resources can not only avoid errors, but also make the code simpler. Note the catch clause may be omitted in a try-with-resources statement.

Note that (1) you have to declare the resource reference variable and create the resource altogether in the `try(...)` clause; (2) the semicolon (`;`) in last statement in the `try(...)` clause may be omitted; (3) You may create multiple `AutoCloseable` resources in the the `try(...)` clause; (4) The `try(...)` clause can contain only the statements for creating resources. Here is an example.


Declare reference to resource
Create resource objects
The ; for the last statement may be omitted

```
try (
  Scanner input = new Scanner(System.in);
  PrintWriter output =
    new PrintWriter("c:\\temp\\temp.txt");
) {
  System.out.println(input.nextLine());
}
```

12.11.3 Reading Data Using **Scanner**

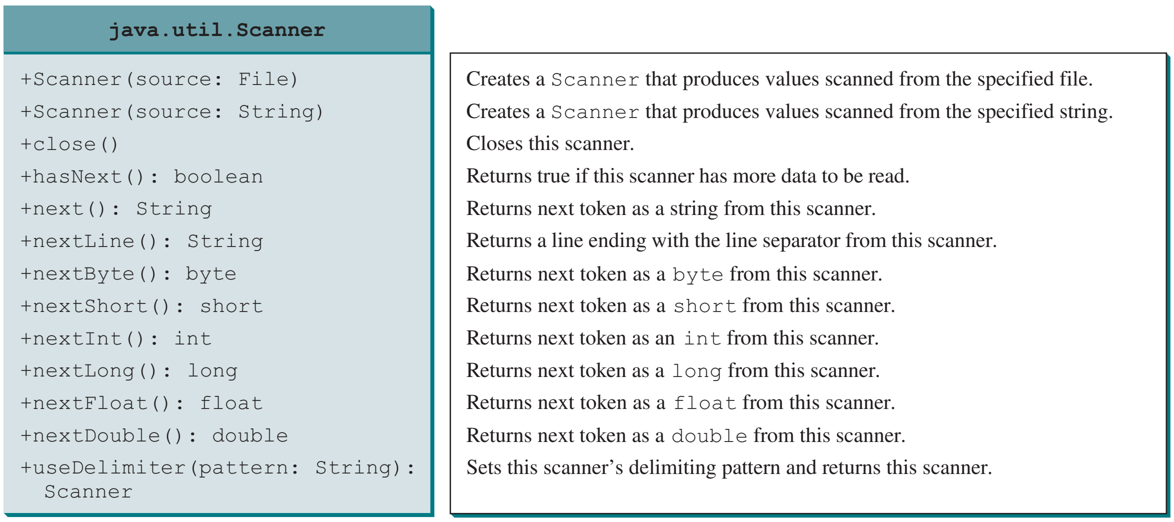
The `java.util.Scanner` class was used to read strings and primitive values from the console in Section 2.3, Reading Input from the Console. A `Scanner` breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a `Scanner` for `System.in`, as follows:

```
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));
```

Figure 12.9 summarizes frequently used methods in `Scanner`.



java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned 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.

Listing 12.15 gives an example that creates an instance of `Scanner` and reads data from the file **scores.txt**.

LISTING 12.15 ReadData.java

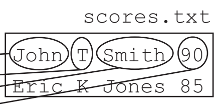
```java
 1 import java.util.Scanner;
 2
 3 public class ReadData {
 4   public static void main(String[] args) throws Exception {
 5     // Create a File instance
 6     java.io.File file = new java.io.File("scores.txt");
 7
 8     // Create a Scanner for the file
 9     Scanner input = new Scanner(file);
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 }
```

create a File

create a Scanner

has next?
read items

close file



File class

Note `new Scanner(String)` creates a `Scanner` for a given string. To create a `Scanner` to read data from a file, you have to use the `java.io.File` class to create an instance of the `File` using the constructor `new File(filename)` (line 6) and use `new Scanner(File)` to create a `Scanner` for the file (line 9).

throws Exception

Invoking the constructor `new Scanner(File)` may throw an I/O exception, so the `main` method declares `throws Exception` in line 4.

Each iteration in the `while` loop reads the first name, middle initial, last name, and score from the text file (lines 12–19). The file is closed in line 22.

close 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. You can rewrite this program using the try-with-resources syntax. See liveexample.pearsoncmg.com/html/ReadDataWithAutoClose.html.

12.11.4 How Does **Scanner** Work?

change delimiter

Section 4.5.5 introduced token-based and line-based input. The token-based input methods **nextByte()**, **nextShort()**, **nextInt()**, **nextLong()**, **nextFloat()**, **nextDouble()**, and **next()** read input separated by delimiters. By default, the delimiters are whitespace characters. You can use the **useDelimiter(String regex)** method to set a new pattern for delimiters.

How does an input method work? A token-based input first skips any delimiters (whitespace characters by default) then reads a token ending at a delimiter. The token is then automatically converted into a value of the **byte**, **short**, **int**, **long**, **float**, or **double** type for **nextByte()**, **nextShort()**, **nextInt()**, **nextLong()**, **nextFloat()**, and **nextDouble()**, respectively. 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.

InputMismatchException

next() vs. nextLine()

Both methods **next()** and **nextLine()** read a string. The **next()** method reads a string separated by delimiters and **nextLine()** reads a line ending with a line separator.

line separator

Note

The line-separator string is defined by the system. It is **\r\n** on Windows and **\n** on UNIX. To get the line separator on a particular platform, use

```
String lineSeparator = System.getProperty("line.separator");
```

If you enter input from a keyboard, a line ends with the *Enter* key, which corresponds to the **\n** character.

behavior of nextLine()

The token-based input method does not read the delimiter after the token. If the **nextLine()** method is invoked after a token-based input 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 the string returned by **nextLine()**.

input from file

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();
```

You will get **34** in **intValue** and an empty string in **line**. Why? Here is the reason. The token-based input method **nextInt()** reads in **34** and stops at the delimiter, which in this case is a line separator (the *Enter* key). The **nextLine()** method ends after reading the line separator and returns the string read before the line separator. Since there are no characters before the line separator, **line** is empty. For this reason, *you should not use a line-based input after a token-based input.*

scan a string

You can read data from a file or from the keyboard using the **Scanner** class. You can also scan data from a string using the **Scanner** class. For example, the following code:

```
Scanner input = new Scanner("13 14");
int sum = input.nextInt() + input.nextInt();
System.out.println("Sum is " + sum);
```

displays

```
Sum is 27
```

12.11.5 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**.

Listing 12.16 gives the program. The program 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).

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);
```

check command usage

source file exists?

target file exists?

try-with-resources

create a `Scanner`

create a `PrintWriter`

has next?
read a line

```
    ) {
      while (input.hasNext()) {
        String s1 = input.nextLine();
        String s2 = s1.replaceAll(args[2], args[3]);
        output.println(s2);
      }
    }
  }
}
```

In a normal situation, the program is terminated after a file is copied. The program is terminated abnormally if the command-line arguments are not used properly (lines 7–11), if the source file does not exist (lines 14–18), or if the target file already exists (lines 22–25). The exit status codes 1, 2, and 3 are used to indicate these abnormal terminations (lines 10, 17, and 24).

Check Point

12.11.1 How do you create a `PrintWriter` to write data to a file? What is the reason to declare `throws Exception` in the main method in Listing 12.13, WriteData.java? What would happen if the `close()` method were not invoked in Listing 12.13?

12.11.2 Show the contents of the file **temp.txt** after the following program is executed:

```
public class Test {
  public static void main(String[] args) throws Exception {
    java.io.PrintWriter output = new
      java.io.PrintWriter("temp.txt");
    output.printf("amount is %f %e\r\n", 32.32, 32.32);
    output.printf("amount is %5.4f %5.4e\r\n", 32.32, 32.32);
    output.printf("%6b\r\n", (1 > 2));
    output.printf("%6s\r\n", "Java");
    output.close();
  }
}
```

12.11.3 Rewrite the code in the preceding question using a try-with-resources syntax.

12.11.4 How do you create a `Scanner` to read data from a file? What is the reason to define `throws Exception` in the main method in Listing 12.15, ReadData.java? What would happen if the `close()` method were not invoked in Listing 12.15?

12.11.5 What will happen if you attempt to create a `Scanner` for a nonexistent file? What will happen if you attempt to create a `PrintWriter` for an existing file?

12.11.6 Is the line separator the same on all platforms? What is the line separator on Windows?

12.11.7 Suppose you enter `45 57.8 789`, then press the *Enter* key. Show the contents of the variables after the following code is executed:

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

12.11.8 Suppose you enter `45`, press the *Enter* key, enter `57.8`, press the *Enter* key, and enter `789`, press the *Enter* key. Show the contents of the variables after the following code is executed:

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