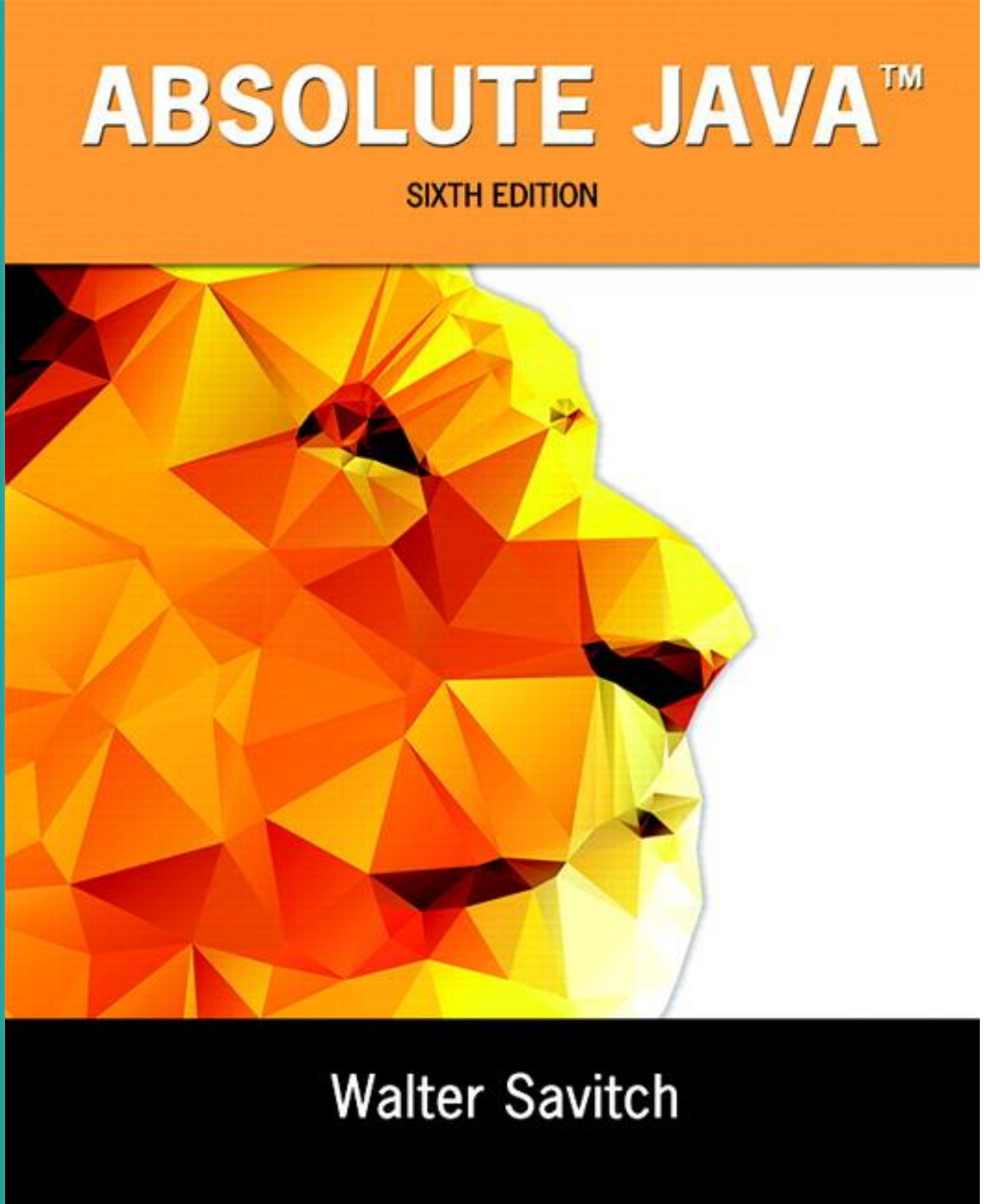

Chapter 10

File I/O

PEARSON

Streams

- A *stream* is an object that enables the flow of data between a program and some I/O device or file
  - If the data flows into a program, then the stream is called an *input stream*
  - If the data flows out of a program, then the stream is called an *output stream*

Streams

- Input streams can flow from the keyboard or from a file
  - **`System.in`** is an input stream that connects to the keyboard

```
Scanner keyboard = new Scanner(System.in);
```

- Output streams can flow to a screen or to a file
  - **`System.out`** is an output stream that connects to the screen

```
System.out.println("Output stream");
```

Text Files and Binary Files

- Files that are designed to be read by human beings, and that can be read or written with an editor are called *text files*
  - Text files can also be called ASCII files because the data they contain uses an ASCII encoding scheme
  - An advantage of text files is that the are usually the same on all computers, so that they can move from one computer to another

Text Files and Binary Files

- Files that are designed to be read by programs and that consist of a sequence of binary digits are called *binary files*
  - Binary files are designed to be read on the same type of computer and with the same programming language as the computer that created the file
  - An advantage of binary files is that they are *more efficient to process* than text files
  - Unlike most binary files, Java binary files have the advantage of being platform independent also

Writing to a Text File

- The class **`PrintWriter`** is a stream class that can be used to write to a text file
  - An object of the class **`PrintWriter`** has the methods **`print`** and **`println`**
  - These are similar to the **`System.out`** methods of the same names, but are used for text file output, not screen output

Writing to a Text File

- All the file I/O classes that follow are in the package **`java.io`**, so a program that uses **`PrintWriter`** will start with a set of **`import`** statements:

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

- The class **`PrintWriter`** has no constructor that takes a file name as its argument
  - It uses another class, **`FileOutputStream`**, to convert a file name to an object that can be used as the argument to its (the **`PrintWriter`**) constructor

Writing to a Text File

- A stream of the class **`PrintWriter`** is created and connected to a text file for writing as follows:

```
PrintWriter outputStreamName;
outputStreamName = new PrintWriter(new
                        FileOutputStream(FileName));
```

  - The class **`FileOutputStream`** takes a string representing the file name as its argument
  - The class **`PrintWriter`** takes the anonymous **`FileOutputStream`** object as its argument

Writing to a Text File

- This produces an object of the class **`PrintWriter`** that is connected to the file ***`FileName`***
  - The process of connecting a stream to a file is called *opening the file*
  - If the file already exists, then doing this causes the old contents to be lost
  - If the file does not exist, then a new, empty file named ***`FileName`*** is created
- After doing this, the methods **`print`** and **`println`** can be used to write to the file

Writing to a Text File

- When a text file is opened in this way, a **`FileNotFoundException`** can be thrown
  - In this context it actually means that the file could not be created
  - This type of exception can also be thrown when a program attempts to open a file for reading and there is no such file
- It is therefore necessary to enclose this code in exception handling blocks
  - The file should be opened inside a **`try`** block
  - A **`catch`** block should catch and handle the possible exception
  - The variable that refers to the **`PrintWriter`** object should be declared outside the block (and initialized to **`null`**) so that it is not local to the block

Writing to a Text File

- When a program is finished writing to a file, it should always close the stream connected to that file

```
outputStreamName.close();
```

  - This allows the system to release any resources used to connect the stream to the file
  - If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

Writing to a Text File

- Output streams connected to files are usually *buffered*
  - Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
  - When enough data accumulates, or when the method **`flush`** is invoked, the buffered data is written to the file all at once
  - This is more efficient, since physical writes to a file can be slow

Writing to a Text File

- The method **`close`** invokes the method **`flush`**, thus insuring that all the data is written to the file
  - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
  - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway
  - The sooner a file is closed after writing to it, the less likely it is that there will be a problem

File Names

- The rules for how file names should be formed depend on a given operating system, not Java
  - When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier (e.g., `"fileName.txt"`)
  - Any suffix used, such as `.txt` has no special meaning to a Java program

A File Has Two Names

- Every input file and every output file used by a program has two names:
  1. The real file name used by the operating system
  2. The name of the stream that is connected to the file
- The actual file name is used to connect to the stream
- The stream name serves as a temporary name for the file, and is the name that is primarily used within the program

IOException

- When performing file I/O there are many situations in which an exception, such as **`FileNotFoundException`**, may be thrown
- Many of these exception classes are subclasses of the class **`IOException`**
  - The class **`IOException`** is the root class for a variety of exception classes having to do with input and/or output
- These exception classes are all checked exceptions
  - Therefore, they must be caught or declared in a throws clause

Unchecked Exceptions

- In contrast, the exception classes **`NoSuchElementException`**, **`InputMismatchException`**, and **`IllegalStateException`** are all unchecked exceptions
  - Unchecked exceptions are not required to be caught or declared in a throws clause

Pitfall: a `try` Block is a Block

- Since opening a file can result in an exception, it should be placed inside a `try` block
- If the variable for a `PrintWriter` object needs to be used outside that block, then the variable must be declared outside the block
  - Otherwise it would be local to the block, and could not be used elsewhere
  - If it were declared in the block and referenced elsewhere, the compiler will generate a message indicating that it is an undefined identifier

Appending to a Text File

- To create a **`PrintWriter`** object and connect it to a text file for *appending*, a second argument, set to **`true`**, must be used in the constructor for the **`FileOutputStream`** object

```
outputStreamName = new PrintWriter(new
  FileOutputStream(FileName, true));
```

  - After this statement, the methods **`print`**, **`println`** and/or **`printf`** can be used to write to the file
  - The new text will be written *after the old text* in the file

`toString` Helps with Text File Output

- If a class has a suitable `toString()` method, and `anObject` is an object of that class, then `anObject` can be used as an argument to `System.out.println`, and it will produce sensible output
- The same thing applies to the methods `print` and `println` of the class `PrintWriter`

```
outputStreamName.println(anObject);
```

Some Methods of the Class `PrintWriter` (Part 1 of 3)

Display 10.2 Some Methods of the Class `PrintWriter`

`PrintWriter` and `FileOutputStream` are in the `java.io` package.

```
public PrintWriter(OutputStream streamObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new PrintWriter(new FileOutputStream(File_Name))
```

When the constructor is used in this way, a blank file is created. If there already was a file named *File_Name*, then the old contents of the file are lost. If you want instead to append new text to the end of the old file contents, use

```
new PrintWriter(new FileOutputStream(File_Name, true))
```

(For an explanation of the argument `true`, read the subsection "Appending to a Text File.")

When used in either of these ways, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream using an object of the class `File`, you can use a `File` object in place of the *File_Name*. (The `File` class will be covered in Section 10.3. We discuss it here so that you will have a more complete reference in this display, but you can ignore the reference to the class `File` until after you've read that section.)

(continued)

Some Methods of the Class `PrintWriter` (Part 2 of 3)

Display 10.2 Some Methods of the Class `PrintWriter`

`public void println(`*Argument*`)`

The *Argument* can be a string, character, integer, floating-point number, `boolean` value, or any combination of these, connected with + signs. The *Argument* can also be any object, although it will not work as desired unless the object has a properly defined `toString()` method. The *Argument* is output to the file connected to the stream. After the *Argument* has been output, the line ends, and so the next output is sent to the next line.

`public void print(`*Argument*`)`

This is the same as `println`, except that this method does not end the line, so the next output will be on the same line.

(continued)

Some Methods of the Class `PrintWriter` (Part 3 of 3)

Display 10.2 Some Methods of the Class `PrintWriter`

`public PrintWriter printf(`*Arguments*`)`

This is the same as `System.out.printf`, except that this method sends output to a text file rather than to the screen. It returns the calling object. However, we have always used `printf` as a `void` method.

`public void close()`

Closes the stream's connection to a file. This method calls `flush` before closing the file.

`public void flush()`

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

Reading From a Text File Using `Scanner`

- The class **`Scanner`** can be used for reading from the keyboard as well as reading from a text file
  - Simply replace the argument **`System.in`** (to the **`Scanner`** constructor) with a suitable stream that is connected to the text file

```
Scanner StreamObject =
    new Scanner(new FileInputStream(FileName));
```

- Methods of the **`Scanner`** class for reading input behave the same whether reading from the keyboard or reading from a text file
  - For example, the **`nextInt`** and **`nextLine`** methods

Reading Input from a Text File Using `Scanner` (Part 1 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4
5  public class TextFileScannerDemo
6  {
7      public static void main(String[] args)
8      {
9         System.out.println("I will read three numbers and a line");
10        System.out.println("of text from the file morestuff.txt.");
11
12        Scanner inputStream = null;
13
14        try
15        {
16            inputStream =
17               new Scanner(new FileInputStream("morestuff.txt"));
18        }
```

(continued)

Reading Input from a Text File Using `Scanner` (Part 2 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
19          catch(FileNotFoundException e)
20          {
21              System.out.println("File morestuff.txt was not found");
22              System.out.println("or could not be opened.");
23              System.exit(0);
24          }
25              int n1 = inputStream.nextInt( );
26              int n2 = inputStream.nextInt( );
27              int n3 = inputStream.nextInt( );
28
29              inputStream.nextLine(); //To go to the next line
30
31              String line = inputStream.nextLine( );
32
```

(continued)

Reading Input from a Text File Using `Scanner` (Part 3 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
33              System.out.println("The three numbers read from the file are:");
34              System.out.println(n1 + ", " + n2 + ", and " +  n3);
35
36              System.out.println("The line read from the file is:");
37              System.out.println(line);
38
39              inputStream.close( );
40          }
41      }
```

File morestuff.txt

```
1 2
3 4
Eat my shorts.
```

This file could have been made with a text editor or by another Java program.

(continued)

Reading Input from a Text File Using `Scanner` (Part 4 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

SCREEN OUTPUT

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

Testing for the End of a Text File with `Scanner`

- A program that tries to read beyond the end of a file using methods of the **`Scanner`** class will cause an exception to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the **`Scanner`** class provides methods such as **`hasNextInt`** and **`hasNextLine`**
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

Checking for the End of a Text File with **`hasNextLine`** (Part 1 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.PrintWriter;
5  import java.io.FileOutputStream;
6
7  public class HasNextLineDemo
8  {
9      public static void main(String[] args)
10     {
11         Scanner inputStream = null;
12         PrintWriter outputStream = null;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 2 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
13          try
14          {
15             inputStream =
16                new Scanner(new FileInputStream("original.txt"));
17             outputStream = new PrintWriter(
18                             new FileOutputStream("numbered.txt"));
19          }
20          catch(FileNotFoundException e)
21          {
22             System.out.println("Problem opening files.");
23             System.exit(0);
24          }

25          String line = null;
26          int count = 0;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 3 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
27          while (inputStream.hasNextLine( ))
28          {
29              line = inputStream.nextLine( );
30              count++;
31              outputStream.println(count + " " + line);
32          }

33          inputStream.close( );
34          outputStream.close( );
35      }

36  }
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 4 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

File original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

File numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

Checking for the End of a Text File with `hasNextInt` (Part 1 of 2)

Display 10.5 Checking for the End of a Text File with `hasNextInt`

```
 1  import java.util.Scanner;
 2  import java.io.FileInputStream;
 3  import java.io.FileNotFoundException;

 4  public class HasNextIntDemo
 5  {
 6      public static void main(String[] args)
 7      {
 8          Scanner inputStream = null;

 9          try
10          {
11             inputStream =
12                new Scanner(new FileInputStream("data.txt"));
13          }
14          catch(FileNotFoundException e)
15          {
16             System.out.println("File data.txt was not found");
17             System.out.println("or could not be opened.");
18             System.exit(0);
19          }
```

(continued)

Checking for the End of a Text File with `hasNextInt` (Part 2 of 2)

Display 10.5 Checking for the End of a Text File with `hasNextInt`

```
20          int next, sum = 0;
21          while (inputStream.hasNextInt( ))
22          {
23              next = inputStream.nextInt( );
24              sum = sum + next;
25          }

26          inputStream.close( );

27          System.out.println("The sum of the numbers is " + sum);
28      }
29  }
```

File data.txt

```
1  2
3  4 hi 5
```

Reading ends when either the end of the file is reach or a token that is not an `int` is reached. So, the `5` is never read.

SCREEN OUTPUT

```
The sum of the numbers is 10
```

Methods in the Class **`Scanner`** (Part 1 of 11)

Display 10.6 Methods in the Class Scanner

`Scanner` is in the `java.util` package.

```
public Scanner(InputStream streamObject)
```

There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you can use

```
new Scanner(new FileInputStream(File_Name))
```

When used in this way, the `FileInputStream` constructor, and thus the `Scanner` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.
To create a stream connected to the keyboard, use
`new Scanner(System.in)`

(continued)

Methods in the Class `Scanner` (Part 2 of 11)

Display 10.6 Methods in the Class Scanner

```
public Scanner(File fileObject)
```

The `File` class will be covered in the section entitled "The `File` Class," later in this chapter. We discuss it here so that you will have a more complete reference in this display, but you can ignore this entry until after you've read that section.
If you want to create a stream using a file name, you can use

```
new Scanner(new File(File_Name))
```

```
public int nextInt()
```

Returns the next token as an `int`, provided the next token is a well-formed string representation of an `int`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of an `int`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 3 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextInt()
```

Returns `true` if the next token is a well-formed string representation of an `int`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public long nextLong()
```

Returns the next token as a `long`, provided the next token is a well-formed string representation of a `long`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `long`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 4 of 11)

Display 10.6 Methods in the Class Scanner

`public boolean hasNextLong()`

Returns `true` if the next token is a well-formed string representation of a `long`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

`public byte nextByte()`

Returns the next token as a `byte`, provided the next token is a well-formed string representation of a `byte`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `byte`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 5 of 11)

Display 10.6 Methods in the Class Scanner

`public boolean hasNextByte()`

Returns `true` if the next token is a well-formed string representation of a `byte`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

`public short nextShort()`

Returns the next token as a `short`, provided the next token is a well-formed string representation of a `short`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `short`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 6 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextShort()
```

Returns `true` if the next token is a well-formed string representation of a `short`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public double nextDouble()
```

Returns the next token as a `double`, provided the next token is a well-formed string representation of a `double`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `double`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 7 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextDouble()
```

Returns `true` if the next token is a well-formed string representation of an `double`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public float nextFloat()
```

Returns the next token as a `float`, provided the next token is a well-formed string representation of a `float`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `float`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 8 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextFloat()
```

Returns `true` if the next token is a well-formed string representation of an `float`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public String next()
```

Returns the `next` token.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 9 of 11)

Display 10.6 Methods in the Class Scanner

`public boolean hasNext()`

Returns `true` if there is another token. May wait for a next token to enter the stream.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

`public boolean nextBoolean()`

Returns the next token as a `boolean` value, provided the next token is a well-formed string representation of a `boolean`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `boolean` value.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class `Scanner` (Part 10 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextBoolean()
```

Returns `true` if the next token is a well-formed string representation of a `boolean` value; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public String nextLine()
```

Returns the rest of the current input line. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.

Throws a `NoSuchElementException` if there are no more lines.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 11 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextLine()
```

Returns `true` if there is a next line. May wait for a next line to enter the stream.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public Scanner useDelimiter(String newDelimiter);
```

Changes the delimiter for input so that `newDelimiter` will be the only delimiter that separates words or numbers. See the subsection "Other Input Delimiters" in Chapter 2 for the details. (You can use this method to set the delimiters to a more complex pattern than just a single string, but we are not covering that.)

Returns the calling object, but we have always used it as a `void` method.

Reading From a Text File Using **BufferedReader**

- The class **BufferedReader** is a stream class that can be used to read from a text file
  - An object of the class **BufferedReader** has the methods **read** and **readLine**
- A program using **BufferedReader**, like one using **PrintWriter**, will start with a set of **import** statements:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

Reading From a Text File Using BufferedReader

- Like the classes **`PrintWriter`** and **`Scanner`**, **`BufferedReader`** has no constructor that takes a file name as its argument
  - It needs to use another class, **`FileReader`**, to convert the file name to an object that can be used as an argument to its (the **`BufferedReader`**) constructor
- A stream of the class **`BufferedReader`** is created and connected to a text file as follows:

```
BufferedReader readerObject;
readerObject = new BufferedReader(new
                        FileReader(FileName));
```

  - This opens the file for reading

Reading From a Text File

- After these statements, the methods **`read`** and **`readLIne`** can be used to read from the file
  - The **`readLine`** method is the same method used to read from the keyboard, but in this case it would read from a file
  - The **`read`** method reads a single character, and returns a value (of type **`int`**) that corresponds to the character read
  - Since the read method does not return the character itself, a type cast must be used:

```
char next = (char)(readerObject.read());
```

Reading Input from a Text File Using **`BufferedReader`** (Part 1 of 3)

Display 10.7 Reading Input from a Text File Using BufferedReader

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;

5  public class TextFileInputDemo
6  {
7      public static void main(String[] args)
8      {
9         try
10        {
11            BufferedReader inputStream =
12               new BufferedReader(new FileReader("morestuff2.txt"));

13            String line = inputStream.readLine();
14            System.out.println(
15                            "The first line read from the file is:");
16            System.out.println(line);
```

(continued)

Reading Input from a Text File Using **`BufferedReader`** (Part 2 of 3)

Display 10.7 Reading Input from a Text File Using BufferedReader

```
17
18              line = inputStream.readLine();
19              System.out.println(
20                              "The second line read from the file is:");
21              System.out.println(line);
22              inputStream.close();
23          }
24          catch(FileNotFoundException e)
25          {
26              System.out.println("File morestuff2.txt was not found");
27              System.out.println("or could not be opened.");
28          }
29          catch(IOException e)
30          {
31              System.out.println("Error reading from morestuff2.txt.");
32          }
33      }
34  }
```

(continued)

Reading Input from a Text File Using **`BufferedReader`** (Part 3 of 3)

Display 10.7 Reading Input from a Text File Using `BufferedReader`

File morestuff2.txt

```
1 2 3
Jack jump over
the candle stick.
```

This file could have been made with a text editor or by another Java program.

SCREEN OUTPUT

```
The first line read from the file is:
1 2 3
The second line read from the file is:
Jack jump over
```

Reading From a Text File

- A program using a **`BufferedReader`** object in this way may throw two kinds of exceptions
  - An attempt to open the file may throw a **`FileNotFoundException`** (which in this case has the expected meaning)
  - An invocation of **`readLine`** may throw an **`IOException`**
  - Both of these exceptions should be handled

Some Methods of the Class `BufferedReader` (Part 1 of 2)

Display 10.8 Some Methods of the Class `BufferedReader`

`BufferedReader` and `FileReader` are in the `java.io` package.

```
public BufferedReader(Reader readerObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new BufferedReader(new FileReader(File_Name))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.
The `File` class will be covered in the section entitled "The `File` Class." We discuss it here so that you will have a more complete reference in this display, but you can ignore the following reference to the class `File` until after you've read that section.
If you want to create a stream using an object of the class `File`, you use

```
new BufferedReader(new FileReader(File_Object))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

(continued)

Some Methods of the Class `BufferedReader` (Part 2 of 2)

Display 10.8 Some Methods of the Class `BufferedReader`

```
public String readLine() throws IOException
```

Reads a line of input from the input stream and returns that line. If the read goes beyond the end of the file, `null` is returned. (Note that an `EOFException` is not thrown at the end of a file. The end of a file is signaled by returning `null`.)

```
public int read() throws IOException
```

Reads a single character from the input stream and returns that character as an `int` value. If the read goes beyond the end of the file, then `-1` is returned. Note that the value is returned as an `int`. To obtain a `char`, you must perform a type cast on the value returned. The end of a file is signaled by returning `-1`. (All of the "real" characters return a positive integer.)

```
public long skip(long n) throws IOException
```

Skips n characters.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

Reading Numbers

- Unlike the **`Scanner`** class, the class **`BufferedReader`** has no methods to read a number from a text file
  - Instead, a number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes
  - To read in a single number on a line by itself, first use the method **`readLine`**, and then use **`Integer.parseInt`**, **`Double.parseDouble`**, etc. to convert the string into a number
  - If there are multiple numbers on a line, **`StringTokenizer`** can be used to decompose the string into tokens, and then the tokens can be converted as described above

Testing for the End of a Text File

- The method **`readLine`** of the class **`BufferedReader`** returns **`null`** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **`null`** when using **`readLine`**
- The method **`read`** of the class **`BufferedReader`** returns **`-1`** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **`-1`** when using **`read`**

Path Names

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given

Path Names

- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists
- A *full path name* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory in which the program is located

Path Names

- The way path names are specified depends on the operating system
  - A typical UNIX path name that could be used as a file name argument is

```
"/user/sallyz/data/data.txt"
```

  - A **BufferedReader** input stream connected to this file is created as follows:

```
BufferedReader inputStream =
  new BufferedReader(new
  FileReader("/user/sallyz/data/data.txt"));
```

Path Names

- The Windows operating system specifies path names in a different way
  - A typical Windows path name is the following:

    ```
    C:\dataFiles\goodData\data.txt
    ```

  - A **BufferedReader** input stream connected to this file is created as follows:

    ```
    BufferedReader inputStream = new
      BufferedReader(new FileReader
      ("C:\\dataFiles\\goodData\\data.txt"));
    ```

  - Note that in Windows **** must be used in place of ****, since a single backslash denotes an the beginning of an escape sequence

Path Names

- A double backslash (\\) must be used for a Windows path name enclosed in a quoted string
  - This problem does not occur with path names read in from the keyboard
- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

Nested Constructor Invocations

- Each of the Java I/O library classes serves only one function, or a small number of functions
  - Normally two or more class constructors are combined to obtain full functionality
- Therefore, expressions with two constructors are common when dealing with Java I/O classes

Nested Constructor Invocations

```
new BufferedReader(new FileReader("stuff.txt"))
```

- Above, the anonymous **`FileReader`** object establishes a connection with the **`stuff.txt`** file
  - However, it provides only very primitive methods for input
- The constructor for **`BufferedReader`** takes this **`FileReader`** object and adds a richer collection of input methods
  - This transforms the inner object into an instance variable of the outer object

`System.in`, `System.out`, and `System.err`

- The standard streams **`System.in`**, **`System.out`**, and **`System.err`** are automatically available to every Java program
  - **`System.out`** is used for normal screen output
  - **`System.err`** is used to output error messages to the screen
- The **`System`** class provides three methods (**`setIn`**, **`setOut`**, and **`setErr`**) for redirecting these standard streams:

```
public static void setIn(InputStream inStream)
public static void setOut(PrintStream outStream)
public static void setErr(PrintStream outStream)
```

`System.in`, `System.out`, and `System.err`

- Using these methods, any of the three standard streams can be redirected
  - For example, instead of appearing on the screen, error messages could be redirected to a file
- In order to redirect a standard stream, a new stream object is created
  - Like other streams created in a program, a stream object used for redirection must be closed after I/O is finished
  - Note, standard streams do not need to be closed

`System.in`, `System.out`, and `System.err`

- Redirecting **`System.err`**:

```
public void getInput()
{
   . . .
   PrintStream errStream = null;
   try
   {
     errStream = new PrintStream(new
           FileOuptputStream("errMessages.txt"));
     System.setErr(errStream);
     . . . //Set up input stream and read
   }
```

System.in, System.out, and System.err

```
   catch(FileNotFoundException e)
   {
      System.err.println("Input file not found");
   }
   finally
   {
      . . .
      errStream.close();
   }
}
```

The `File` Class

- The **`File`** class is like a wrapper class for file names
  - The constructor for the class **`File`** takes a name, (known as the *abstract name)* as a string argument, and produces an object that represents the file with that name
  - The **`File`** object and methods of the class **`File`** can be used to determine information about the file and its properties

Some Methods in the Class `File` (Part 1 of 5)

Display 10.12 Some Methods in the Class `File`

`File` is in the `java.io` package.

```
public File(String File_Name)
```

Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.

```
public boolean exists()
```

Tests whether there is a file with the abstract path name.

```
public boolean canRead()
```

Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable by the program; otherwise returns `false`.

(continued)

Some Methods in the Class `File` (Part 2 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean setReadOnly()
```

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

```
public boolean canWrite()
```

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

```
public boolean delete()
```

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

(continued)

Some Methods in the Class `File` (Part 3 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns `true` if successful, and returns `false` otherwise.

```
public String getName()
```

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract path name as a `String` value.

```
public boolean renameTo(File New_Name)
```

Renames the file represented by the abstract path name to *New_Name*. Returns `true` if successful; otherwise returns `false`. *New_Name* can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

(continued)

Some Methods in the Class `File` (Part 4 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean isFile()
```

Returns `true` if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns `false`. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

```
public boolean isDirectory()
```

Returns `true` if a directory (folder) exists that is named by the abstract path name; otherwise returns `false`.

(continued)

Some Methods in the Class `File` (Part 5 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean mkdir()
```

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

```
public boolean mkdirs()
```

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

```
public long length()
```

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.