OBJECT-ORIENTED SYSTEMS DESIGN: File I/O

2022/05/16

Department of Computer Science,
Hanyang University

Taeuk Kim

Introduction to File I/O

Chapter 10.1

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*.

- **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

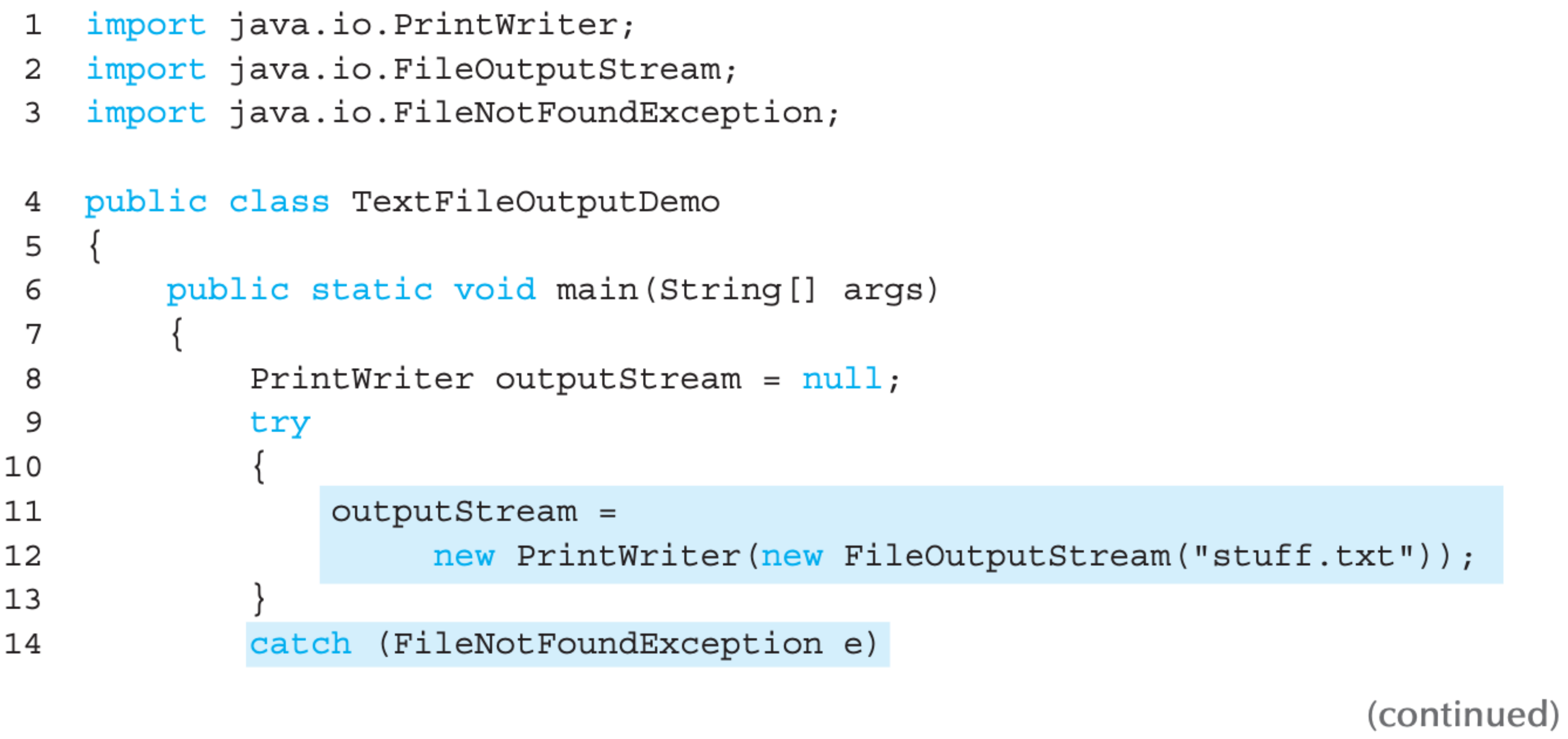
- **Files that are designed to be read by human beings, and that can be read or written with an editor are called *text files.***
  - Text files can also be called ASCII files because the data they contain uses an ASCII encoding scheme.
  - An advantage of text files is that the are usually the same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

- **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.

Text Files

Chapter 10.2

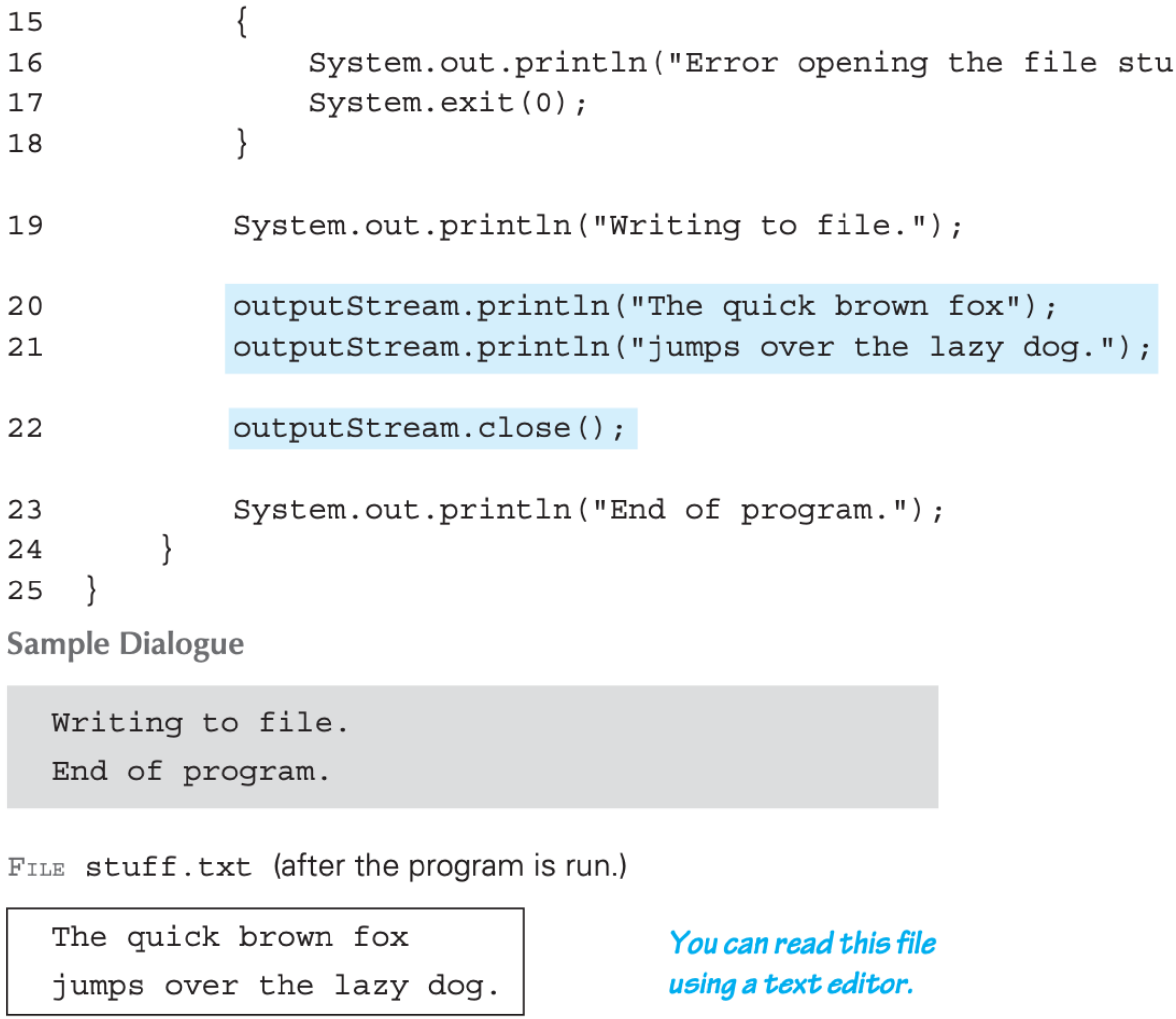
Writing to a Text File



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

public class TextFileOutputDemo
{
    public static void main(String[] args)
    {
        PrintWriter outputStream = null;
        try
        {
            outputStream =
                 new PrintWriter(new FileOutputStream("stuff.txt"));
        }
        catch (FileNotFoundException e)
```

(continued)

Writing to a Text File



```
        {
            System.out.println("Error opening the file stuff.txt.");
            System.exit(0);
        }

        System.out.println("Writing to file.");

        outputStream.println("The quick brown fox");
        outputStream.println("jumps over the lazy dog.");

        outputStream.close();

        System.out.println("End of program.");
    }
}
```

Sample Dialogue

```
Writing to file.
End of program.
```

FILE stuff.txt (after the program is run.)

```
The quick brown fox
jumps over the lazy dog.
```

You can read this file using a text editor.

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.

- **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.

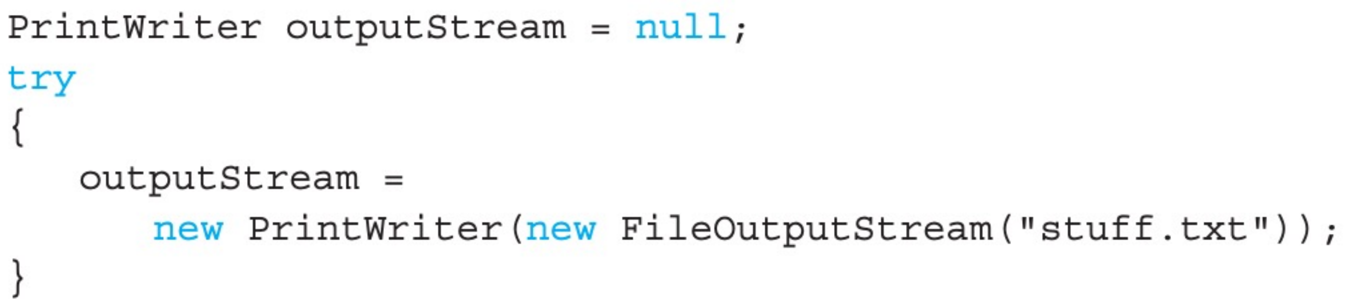
Writing to a Text File

- **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.

- **The method `close` invokes the method `flush`, thus ensuring 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.

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.



```
PrintWriter outputStream = null;
try
{
    outputStream =
        new PrintWriter(new FileOutputStream("stuff.txt"));
}
```

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.

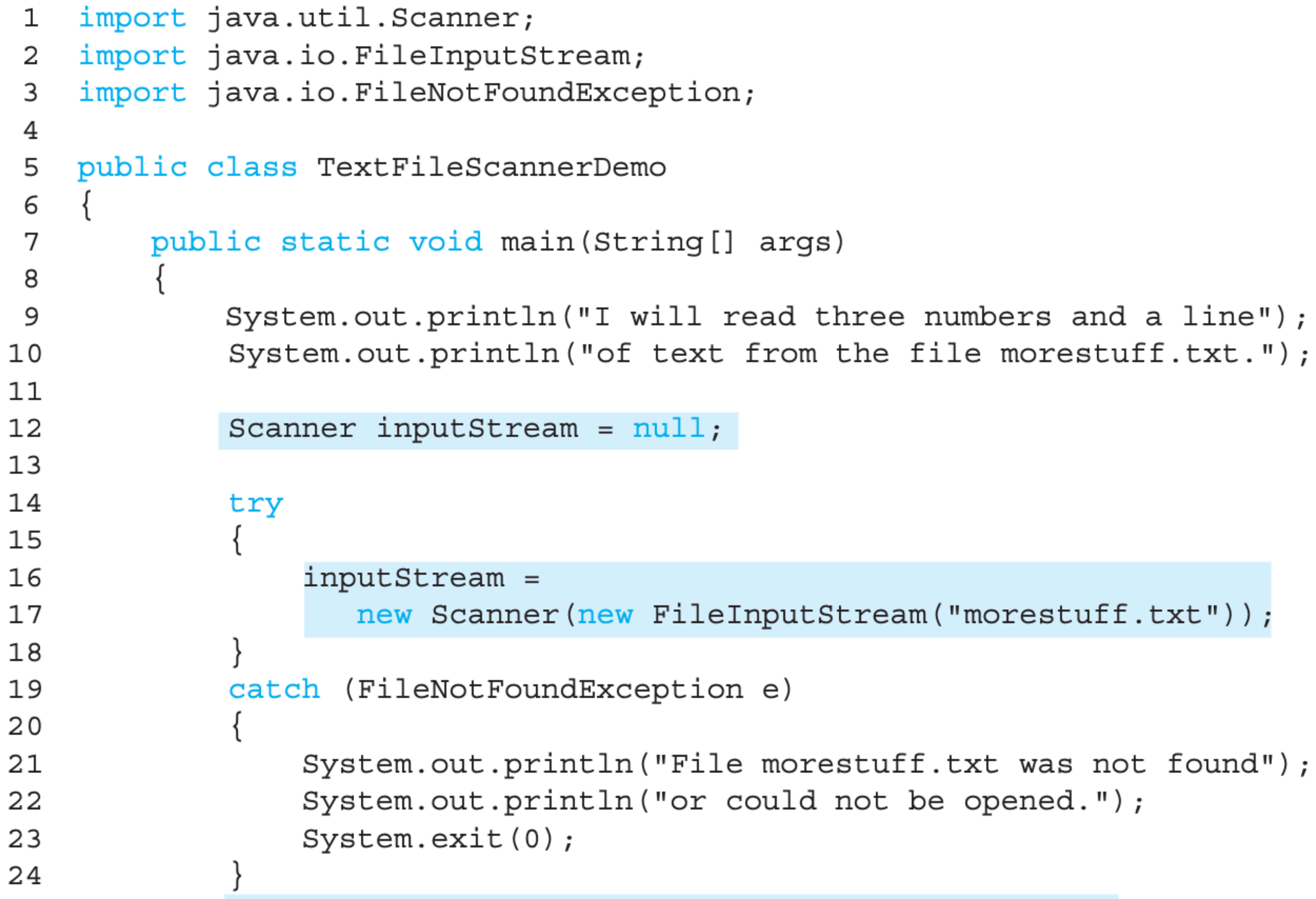
Reading From a Text File Using `Scanner`

- **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));
```

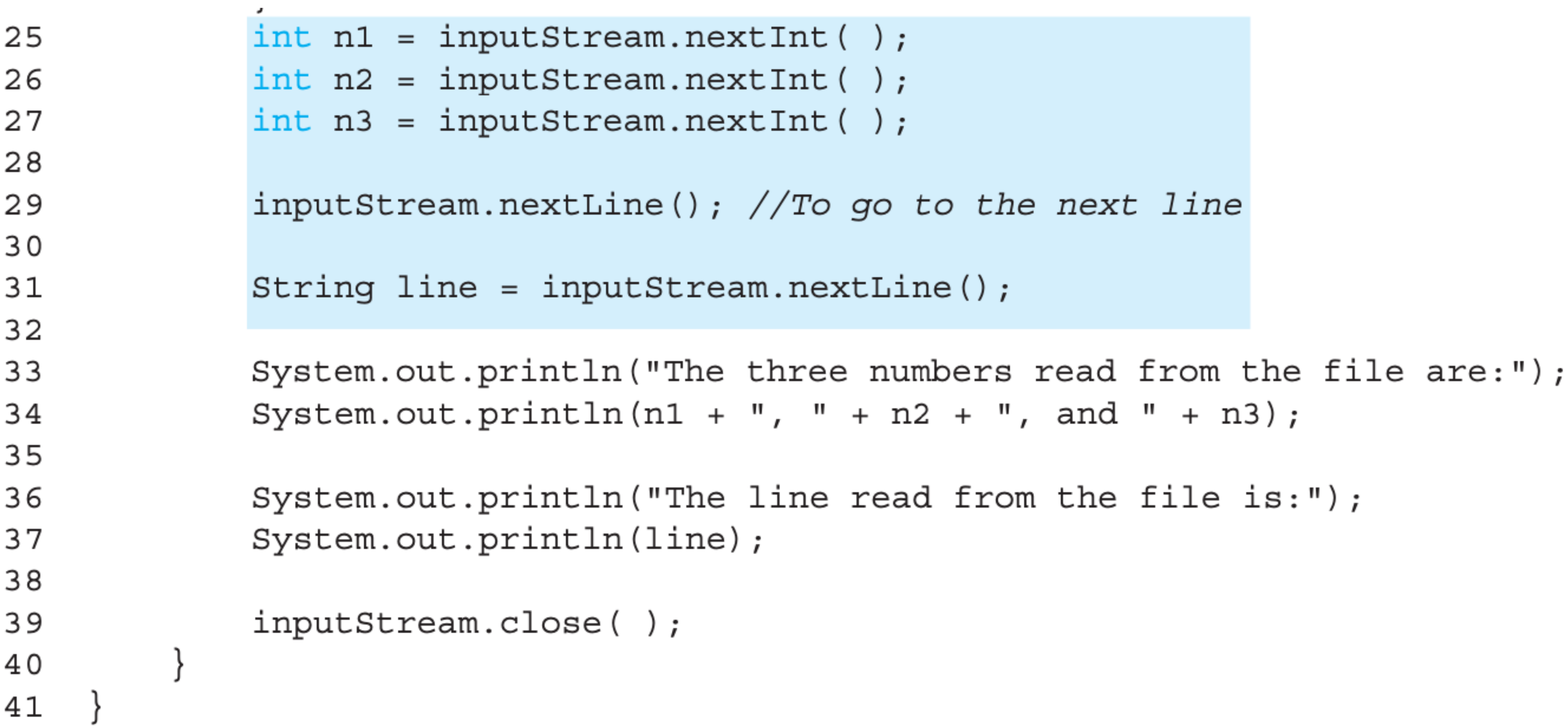
- **Methods of the `Scanner` class for reading input behave the same whether reading from the keyboard or reading from a text file.**
  - For example, the `nextInt` and `nextLine` methods.

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

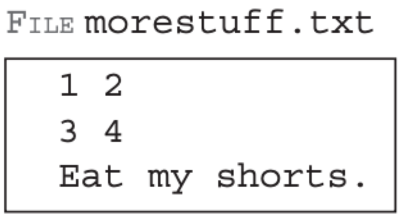


```java
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class TextFileScannerDemo
{
    public static void main(String[] args)
    {
        System.out.println("I will read three numbers and a line");
        System.out.println("of text from the file morestuff.txt.");

        Scanner inputStream = null;

        try
        {
            inputStream =
               new Scanner(new FileInputStream("morestuff.txt"));
        }
        catch (FileNotFoundException e)
        {
            System.out.println("File morestuff.txt was not found");
            System.out.println("or could not be opened.");
            System.exit(0);
        }
```

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



```
        int n1 = inputStream.nextInt( );
        int n2 = inputStream.nextInt( );
        int n3 = inputStream.nextInt( );

        inputStream.nextLine(); //To go to the next line

        String line = inputStream.nextLine();

        System.out.println("The three numbers read from the file are:");
        System.out.println(n1 + ", " + n2 + ", and " + n3);

        System.out.println("The line read from the file is:");
        System.out.println(line);

        inputStream.close( );
    }
}
```



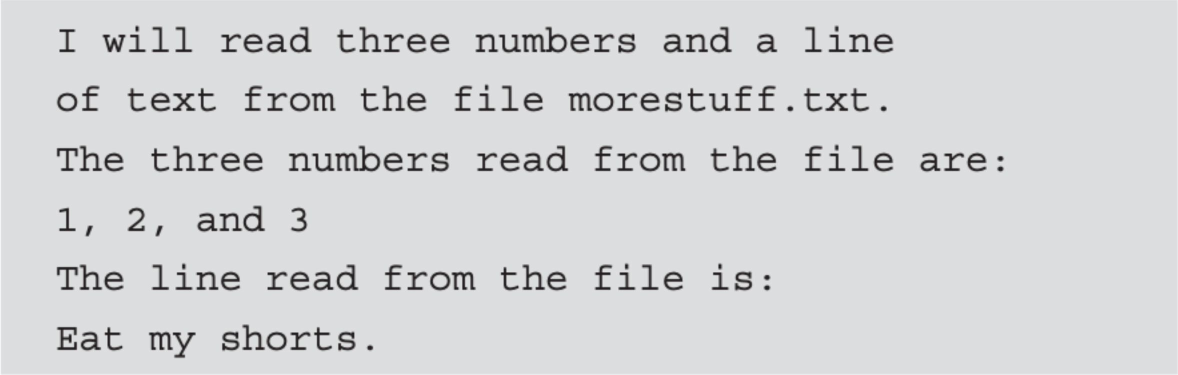
FILE morestuff.txt

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

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

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

Screen Output



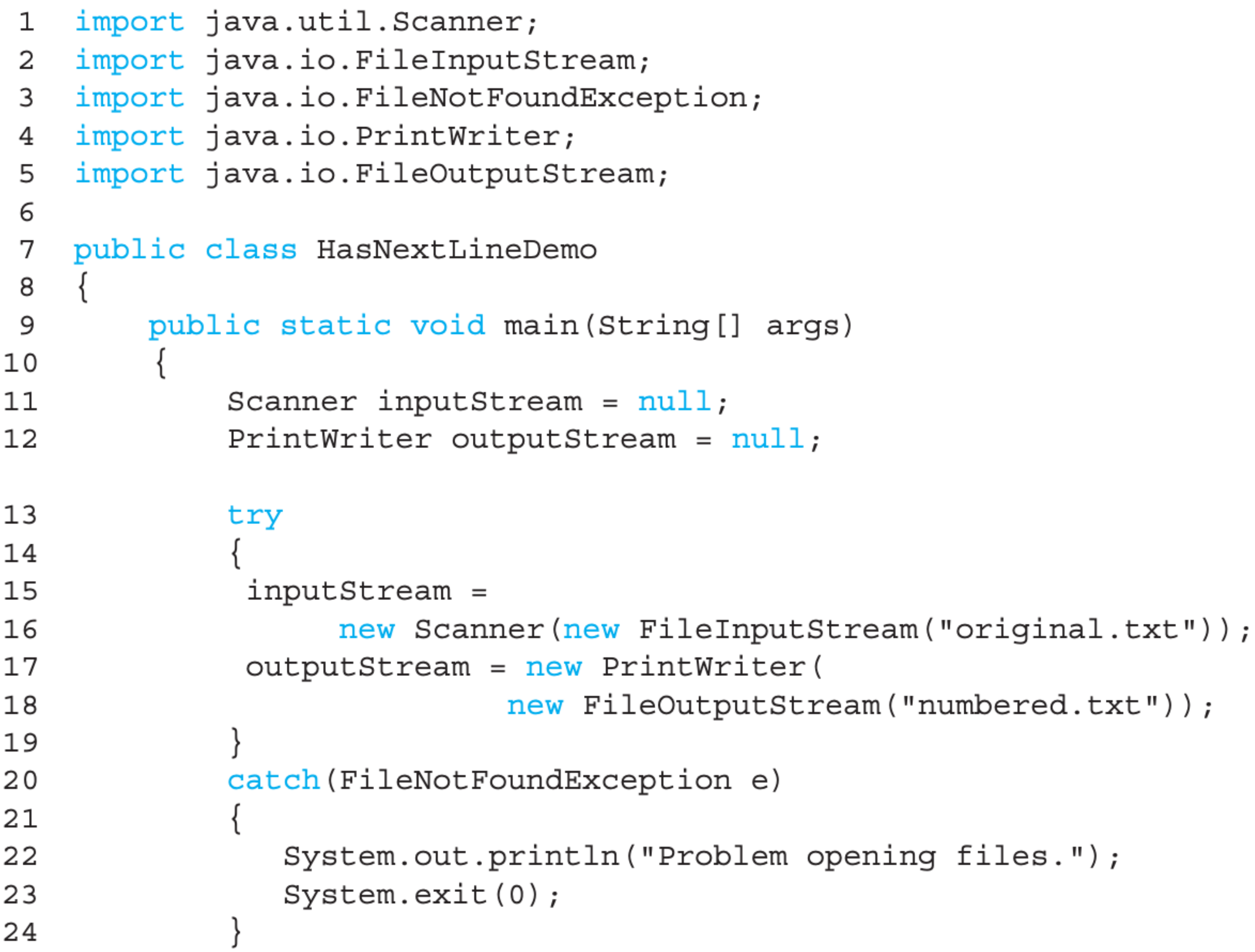
```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

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.**

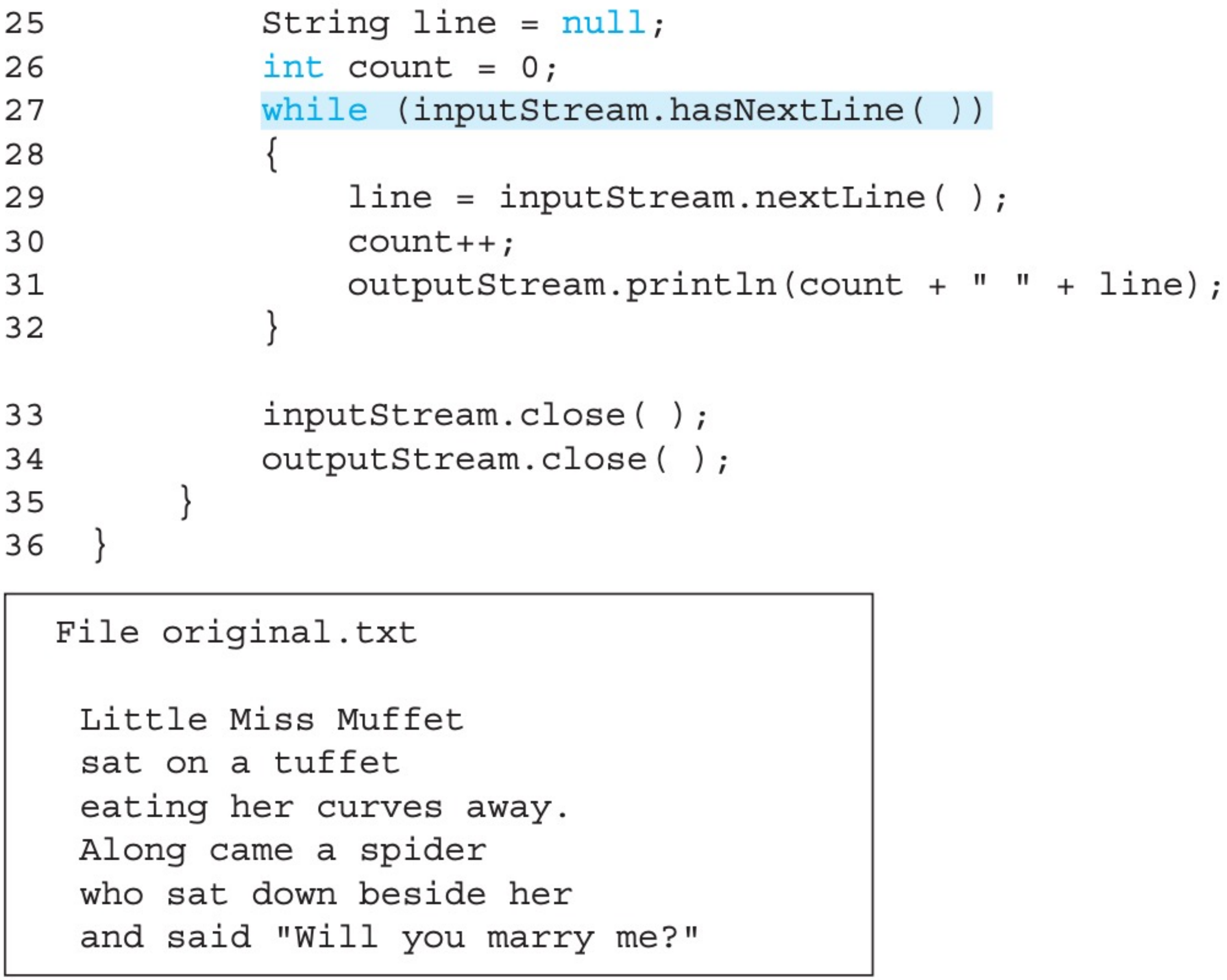
- **However, instead of having to rely on an exception to signal the end of a file, the `Scanner` class provides methods such as `hasNextInt` and `hasNextLine`.**
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type.

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



```java
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.FileOutputStream;

public class HasNextLineDemo
{
    public static void main(String[] args)
    {
        Scanner inputStream = null;
        PrintWriter outputStream = null;

        try
        {
         inputStream =
              new Scanner(new FileInputStream("original.txt"));
         outputStream = new PrintWriter(
                         new FileOutputStream("numbered.txt"));
        }
        catch(FileNotFoundException e)
        {
           System.out.println("Problem opening files.");
           System.exit(0);
        }
```

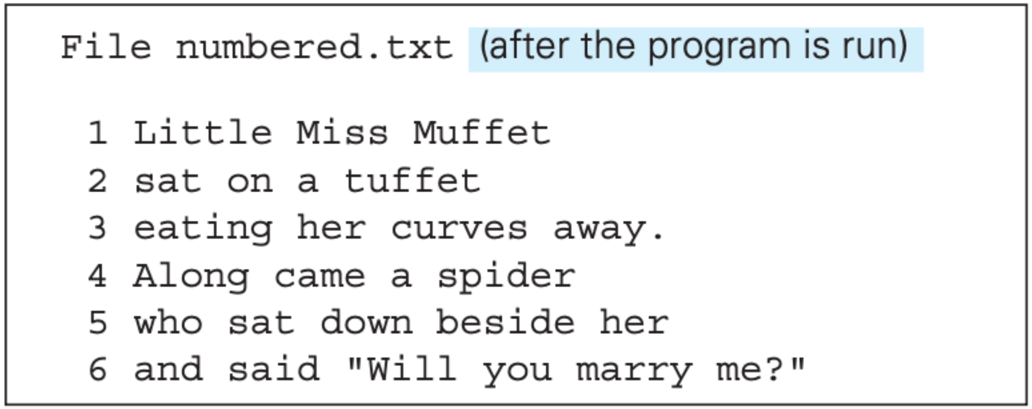
Checking for the End of a Text File with `hasNextLine`



```
        String line = null;
        int count = 0;
        while (inputStream.hasNextLine( ))
        {
            line = inputStream.nextLine( );
            count++;
            outputStream.println(count + " " + line);
        }

        inputStream.close( );
        outputStream.close( );
    }
}
```

```
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?"
```

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



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?"
```

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));
```

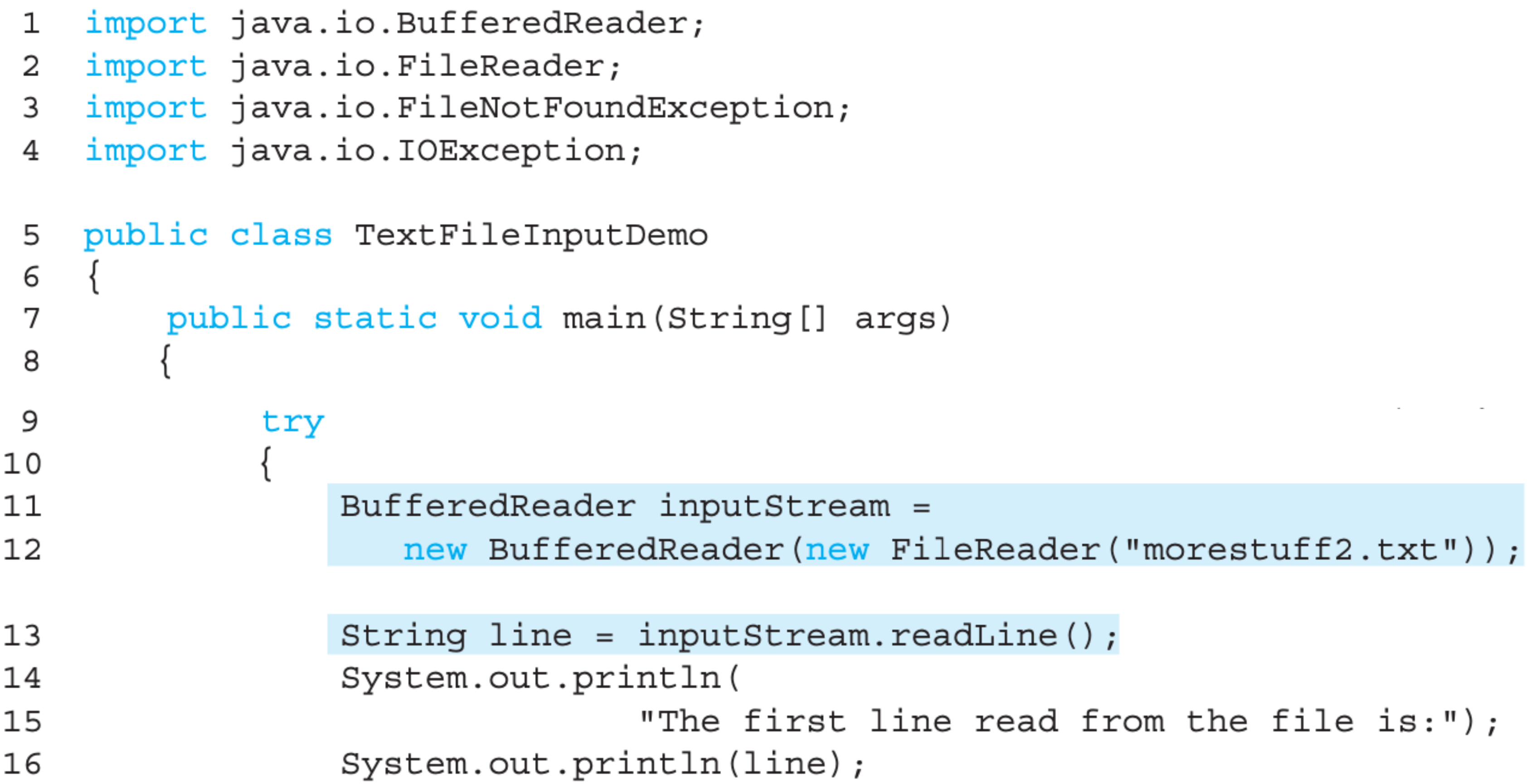
  - This opens the file for reading.

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:

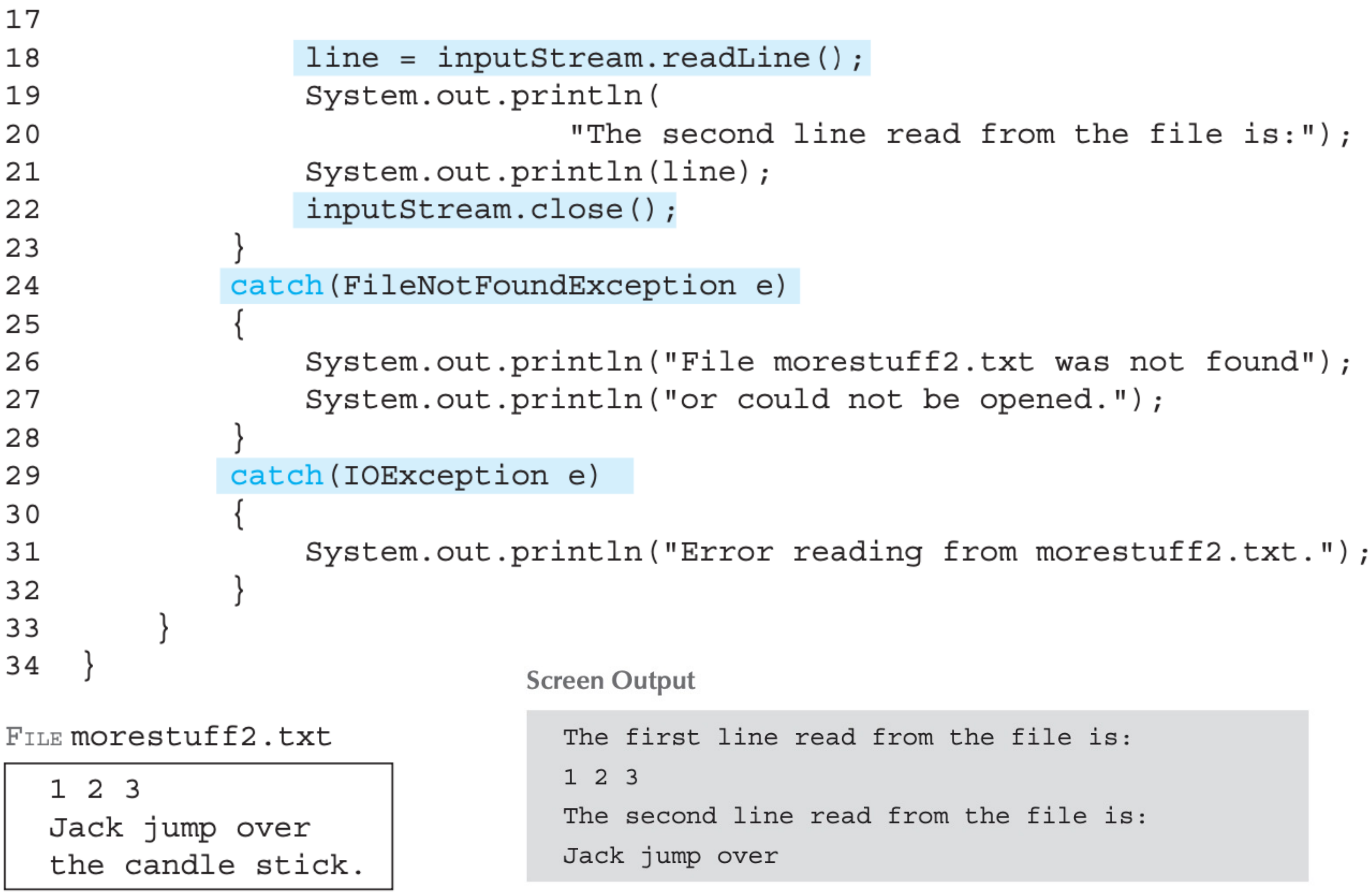
```
char next = (char)(readerObject.read());
```

Reading Input from a Text File Using `BufferedReader`



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

public class TextFileInputDemo
{
    public static void main(String[] args)
    {
        try
        {
            BufferedReader inputStream =
               new BufferedReader(new FileReader("morestuff2.txt"));

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

Reading Input from a Text File Using BufferedReader



```
            line = inputStream.readLine();
            System.out.println(
                          "The second line read from the file is:");
            System.out.println(line);
            inputStream.close();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("File morestuff2.txt was not found");
            System.out.println("or could not be opened.");
        }
        catch(IOException e)
        {
            System.out.println("Error reading from morestuff2.txt.");
        }
    }
}
```

File morestuff2.txt

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

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 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.

- 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.

Summary: 10. File I/O

- **10.1 Introduction to File I/O**
  - Streams
  - Text Files and Binary Files
- **10.2 Text Files**
  - Writing/appending to a Text File
  - Reading from a Text File
  - Reading a Text File Using **`Scanner`**
  - Reading a Text File Using **`BufferedReader`**
  - PathNames

OBJECT-ORIENTED SYSTEMS DESIGN: Recursion

2022/05/16

Department of Computer Science,
Hanyang University

Taeuk Kim

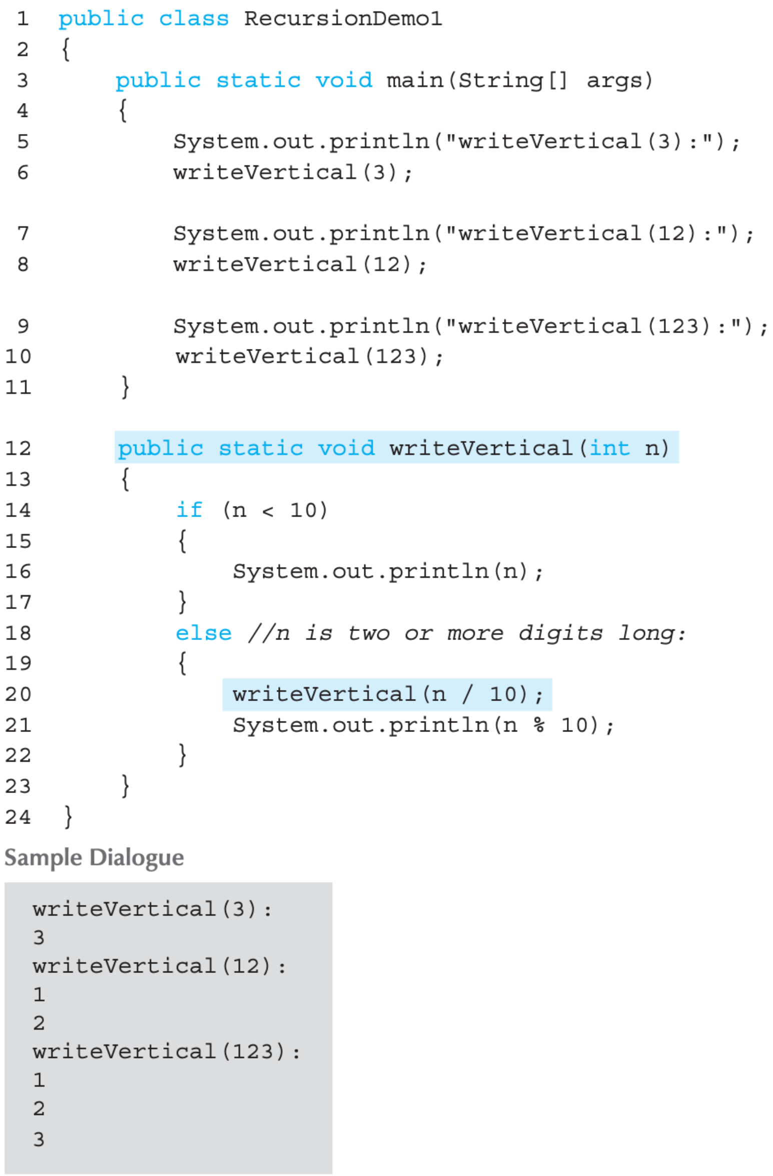
Recursive `Void` Methds

Chapter 11.1

Recursive Methods

- **A *recursive* method is a method that includes a call to itself.**
- **Recursion is based on the general problem solving technique of breaking down a task into subtasks.**
  - In particular, recursion can be used whenever one subtask is a smaller version of the original task.

A Recursive `void` Method



```java
public class RecursionDemo1
{
    public static void main(String[] args)
    {
        System.out.println("writeVertical(3):");
        writeVertical(3);

        System.out.println("writeVertical(12):");
        writeVertical(12);

        System.out.println("writeVertical(123):");
        writeVertical(123);
    }

    public static void writeVertical(int n)
    {
        if (n < 10)
        {
            System.out.println(n);
        }
        else //n is two or more digits long:
        {
            writeVertical(n / 10);
            System.out.println(n % 10);
        }
    }
}
```

Sample Dialogue

```
writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3
```

A Closer Look at Recursion

- **The computer keeps track of recursive calls as follows:**
  - When a method is called, the computer plugs in the arguments for the parameter(s), and starts executing the code.
  - If it encounters a recursive call, it temporarily stops its computation.
  - When the recursive call is completed, the computer returns to finish the outer computation.

- **When the computer encounters a recursive call, it must temporarily suspend its execution of a method.**
  - It does this because *it must know the result of the recursive call before it can proceed*.
  - It saves all the information it needs to continue the computation later on, when it returns from the recursive call.

- **Ultimately, this entire process terminates when one of the recursive calls does not depend upon recursion to return.**

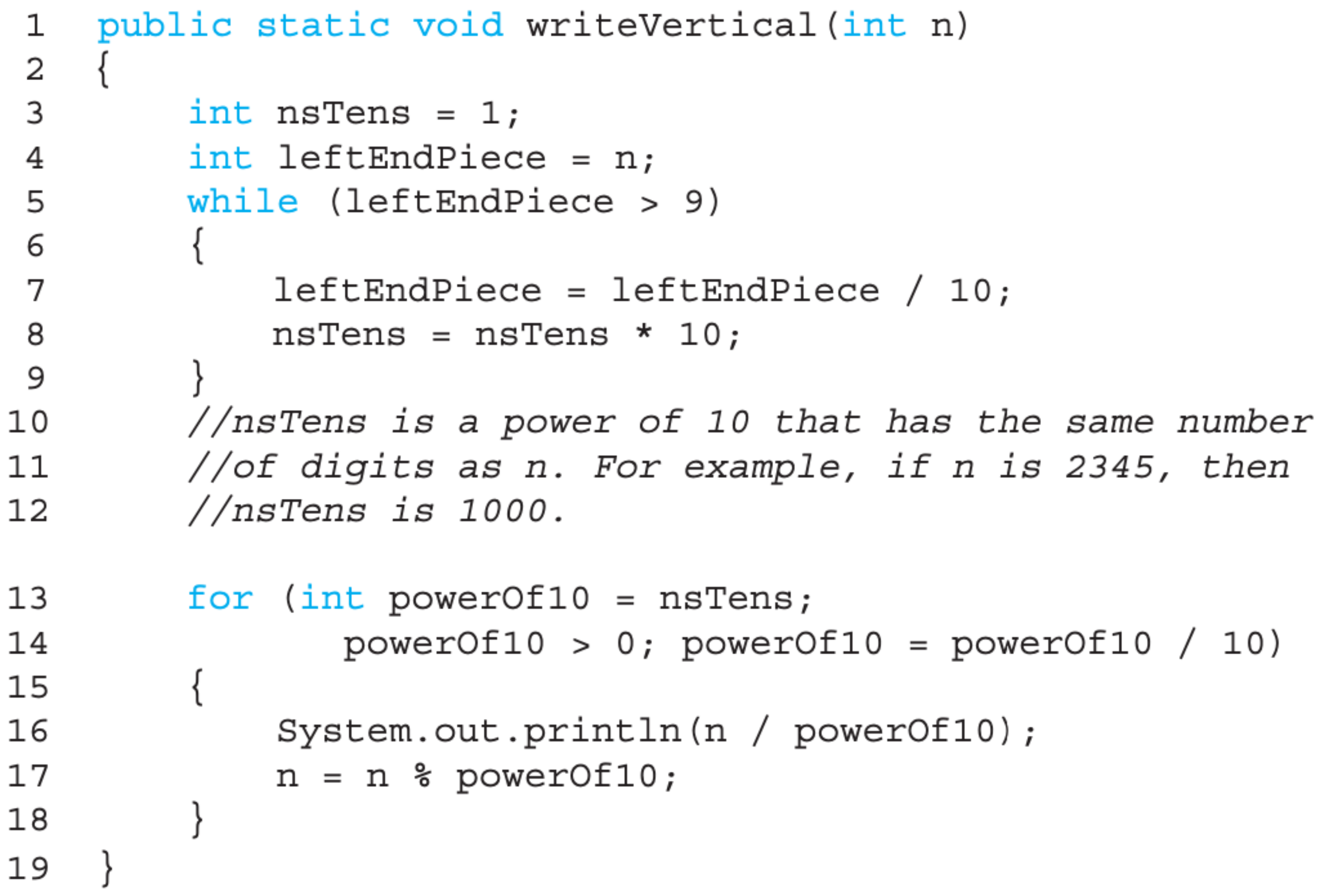
Recursion Versus Iteration

- **Recursion is not absolutely necessary**
  - Any task that can be done using recursion can also be done in a nonrecursive manner.
  - A nonrecursive version of a method is called an *iterative version.*

- **An iteratively written method will typically use loops of some sort in place of recursion.**

- **A recursively written method can be simpler, but will usually run slower and use more storage than an equivalent iterative version.**

Iterative version of `writeVertical`



```java
public static void writeVertical(int n)
{
    int nsTens = 1;
    int leftEndPiece = n;
    while (leftEndPiece > 9)
    {
        leftEndPiece = leftEndPiece / 10;
        nsTens = nsTens * 10;
    }
    //nsTens is a power of 10 that has the same number
    //of digits as n. For example, if n is 2345, then
    //nsTens is 1000.

    for (int powerOf10 = nsTens;
           powerOf10 > 0; powerOf10 = powerOf10 / 10)
    {
        System.out.println(n / powerOf10);
        n = n % powerOf10;
    }
}
```

Recursive Methods That Return a Value

Chapter 11.2

Recursive Methods that Return a Value

- **Recursion is not limited to `void` methods.**

- **A recursive method can return a value of any type.**

- **An outline for a successful recursive method that returns a value is as follows:**
  - One or more cases in which the value returned is computed in terms of calls to the same method.
  - the arguments for the recursive calls should be intuitively "smaller”.
  - One or more cases in which the value returned is computed without the use of any recursive calls (the base or stopping cases).

Another Powers Method

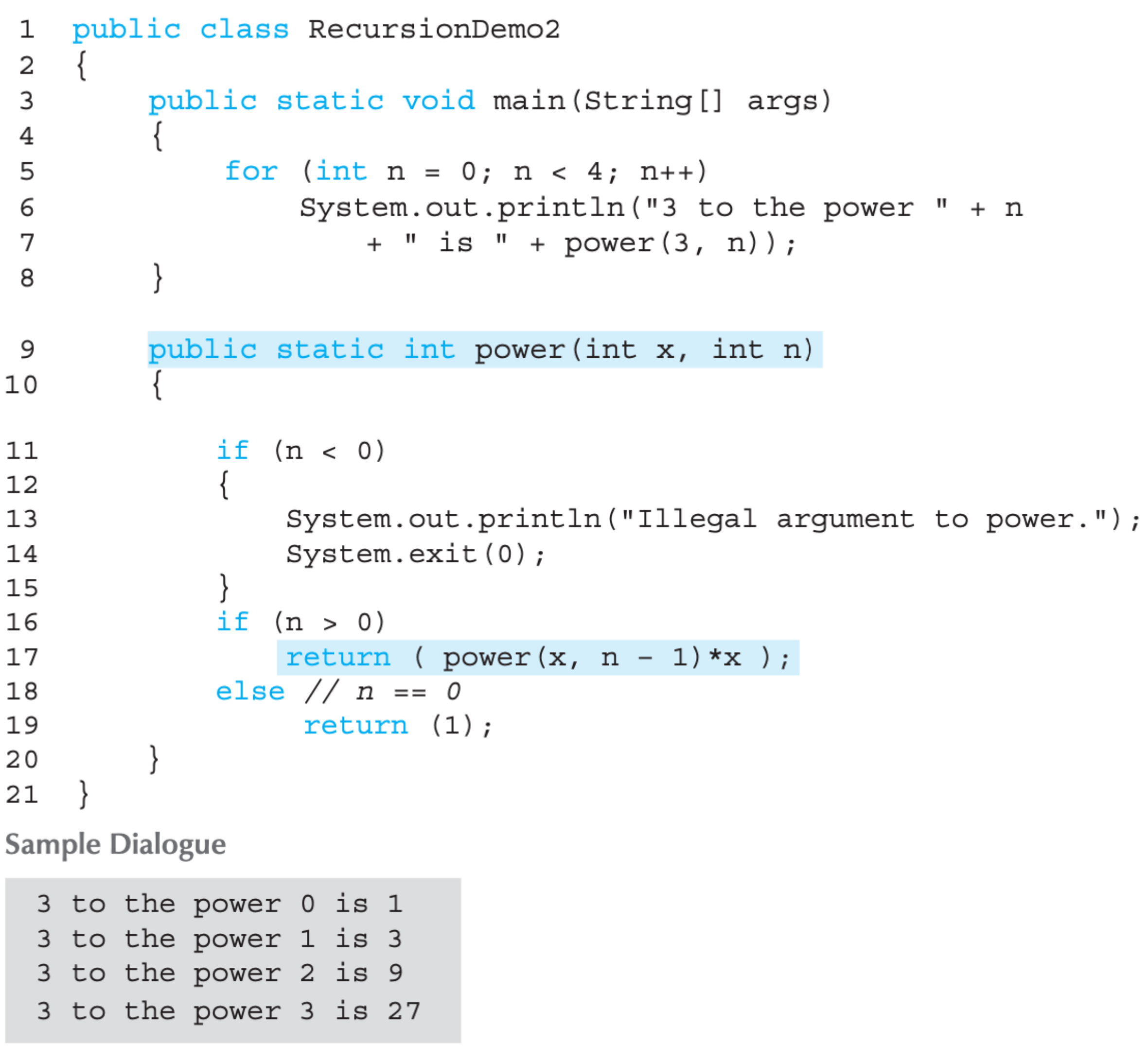
- **The method `pow` from the Math class computes powers.**
  - It takes two arguments of type `double` and returns a value of type `double`.

- **The recursive method `power` takes two arguments of type `int` and returns a value of type `int`.**
  - The definition of `power` is based on the following formula:
  - **x^n is equal to $x^{n-1} * x$**

- **In terms of Java, the value returned by `power(x, n)` for `n>0` should be the same as**
  - `power(x, n-1) * x.`

- **When `n=0`, then `power(x, n)` should return `1`**
  - This is the stopping case.

The Recursive Method `power` (Part 1 of 2)

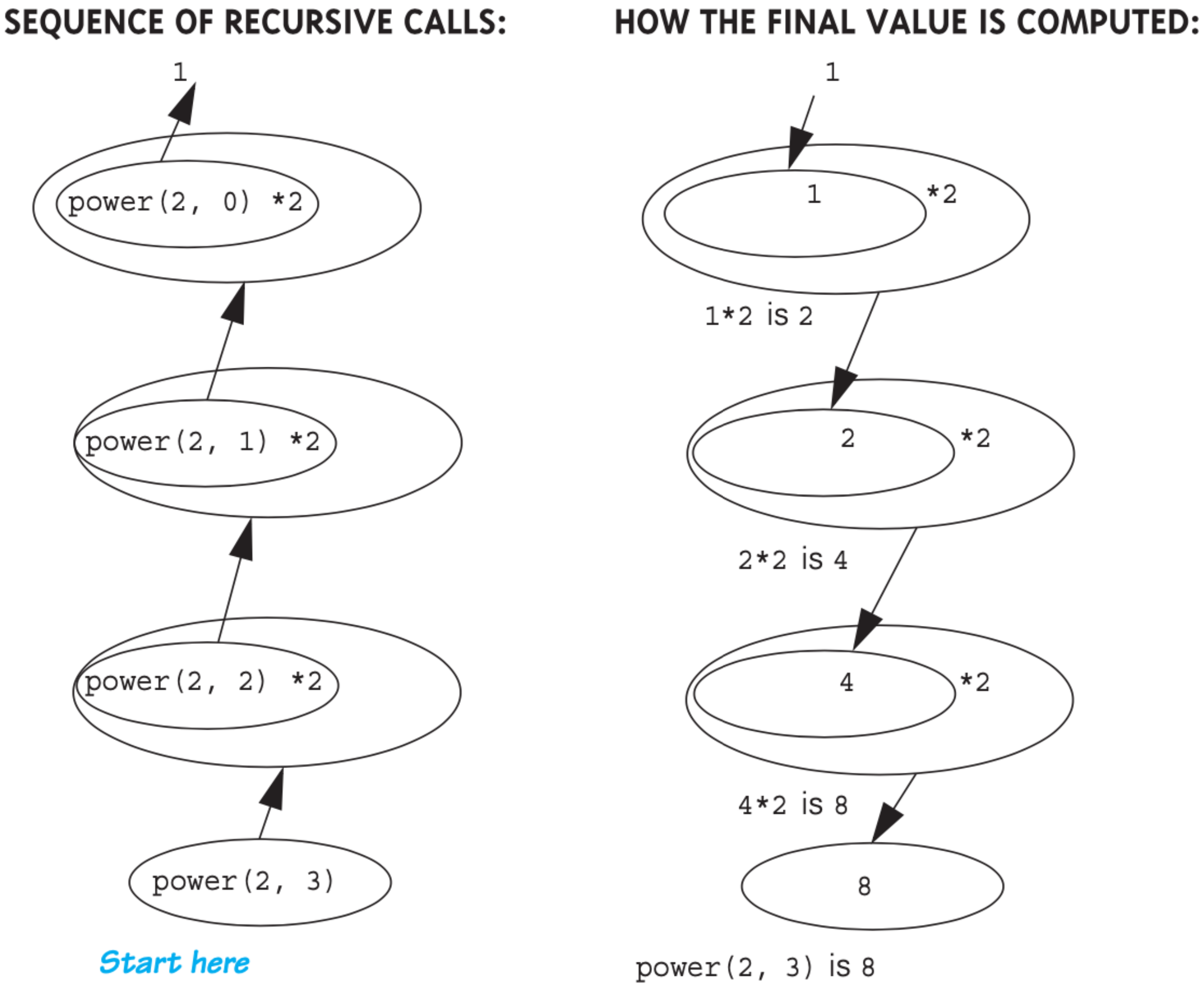


```java
public class RecursionDemo2
{
    public static void main(String[] args)
    {
        for (int n = 0; n < 4; n++)
            System.out.println("3 to the power " + n
                + " is " + power(3, n));
    }

    public static int power(int x, int n)
    {

        if (n < 0)
        {
            System.out.println("Illegal argument to power.");
            System.exit(0);
        }
        if (n > 0)
            return ( power(x, n - 1)*x );
        else // n == 0
             return (1);
    }
}
```

Sample Dialogue

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

The Recursive Method `power` (Part 2 of 2)



Thinking Recursively

Chapter 11.3

Thinking Recursively

- **If a problem lends itself to recursion, it is more important to think of it in recursive terms.**

- **In the case of methods that return a value, there are three properties that must be satisfied, as follows:**
  - There is no infinite recursion: every chain of recursive calls must reach a stopping case.
  - Each stopping case returns the correct value for that case.
  - For the cases that involve recursion: *if* all recursive calls return the correct value, *then* the final value returned by the method is the correct value.

- **These properties follow a technique also known as *mathematical induction.***

Summary: 11. Recursion

- **11.1 Recursive `void` Methods**
- **11.2 Recursive Methods That Returan a Value**
- **11.3 Thinking Recursively**

Next Lecture

- 12. UML and Patterns

- 13. Interfaces and Inner Classes

Q & A