Chapter 10 File I/O

1. Stream: an object that enables the flow of data between a program and some I/O device or file
   a. input stream: data flows into a program
      i. System.in: connects to keyboard
      ```
      Scanner keyboard = new Scanner(System.in);
      ```
   b. output stream: data flows out of a program
      i. System.out: connects to screen
      ```
      System.out.println(Output stream‟);
      ```
2. Text Files(“ASCII files”): readable by human
3. Binary Files: sequence of binary digits
   a. more efficient than text files
4. Writing to a Text File: `class PrintWriter`
   a. stream class that can write to a text file
   b. has `print` and `println`
   c. need to import the followings:
   ```
   import java.io.PrintWriter;
   import java.io.FileOutputStream;
   import java.io.FileNotFoundException;
   ```
   d. `PrintWrite` has no constructor to take file name, so uses `FileOutputStream` to convert a file name to an object that can be used as the argument to `PrintWriter` constructor
   e. for example
   ```
   /* 1. Open file.
      2. If file already exists, replace it.
      3. If file doesn’t exist, create it.
      4. nested constructor invocations on right hand side*/
   PrintWriter outputStreamName = new PrintWriter(new
                                  FileOutputStream(FileName));
   ```

   f. close the stream when finish
      i. `outputStreamName.close();`
      ii. release any resources used to connect the stream to the file
      iii. Java automatically closes it when program ends.
      iv. Programmer should explicitly close it.

   g. streams are buffered
      i. data is saved in a temporary location (buffer) instead of writing to the file ASAP
         1. Because I/O devices are slow. 累積一定程度再一次輸出比較有效率
      ii. buffered data is written to file all at once, when

            1. enough data accumulates
            2. method `flush` is invoked: insuring all data is written to the file

5. file name
   a. suffix(`.txt`, `.exe`, ...) no meaning to Java program
   b. every input file and output file used by program has two names
      i. real file name (used by the operating system)
      ii. stream name (connected to the file)

6. IOException
   a. root class for input/output exceptions, e.g. `FileNotFoundException`
   b. all are checked exceptions (must be caught)
7. Unchecked Exception
   a. not required to be caught
   b. for example, `NoSuchElementException`, `InputMismatchException`, and `IllegalStateException`
8. Appending to a Text File
   a. syntax:

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

9. `toString` helps with Text File Output
   a. if a class has `toString()` method, it can be used as an argument to `System.out.println` directly

```
//no anObject.toString() required
outputStreamName.println(anObject)
```

10. Reading from a text file: Scanner
    a. syntax:

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

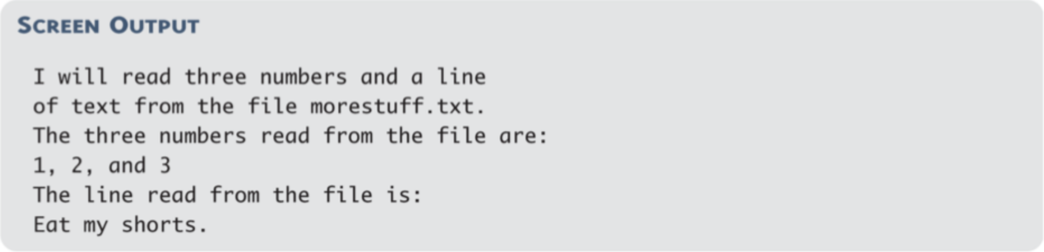
    b. has `nextInt` and `nextLine` methods

```
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);
       }
           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.
```

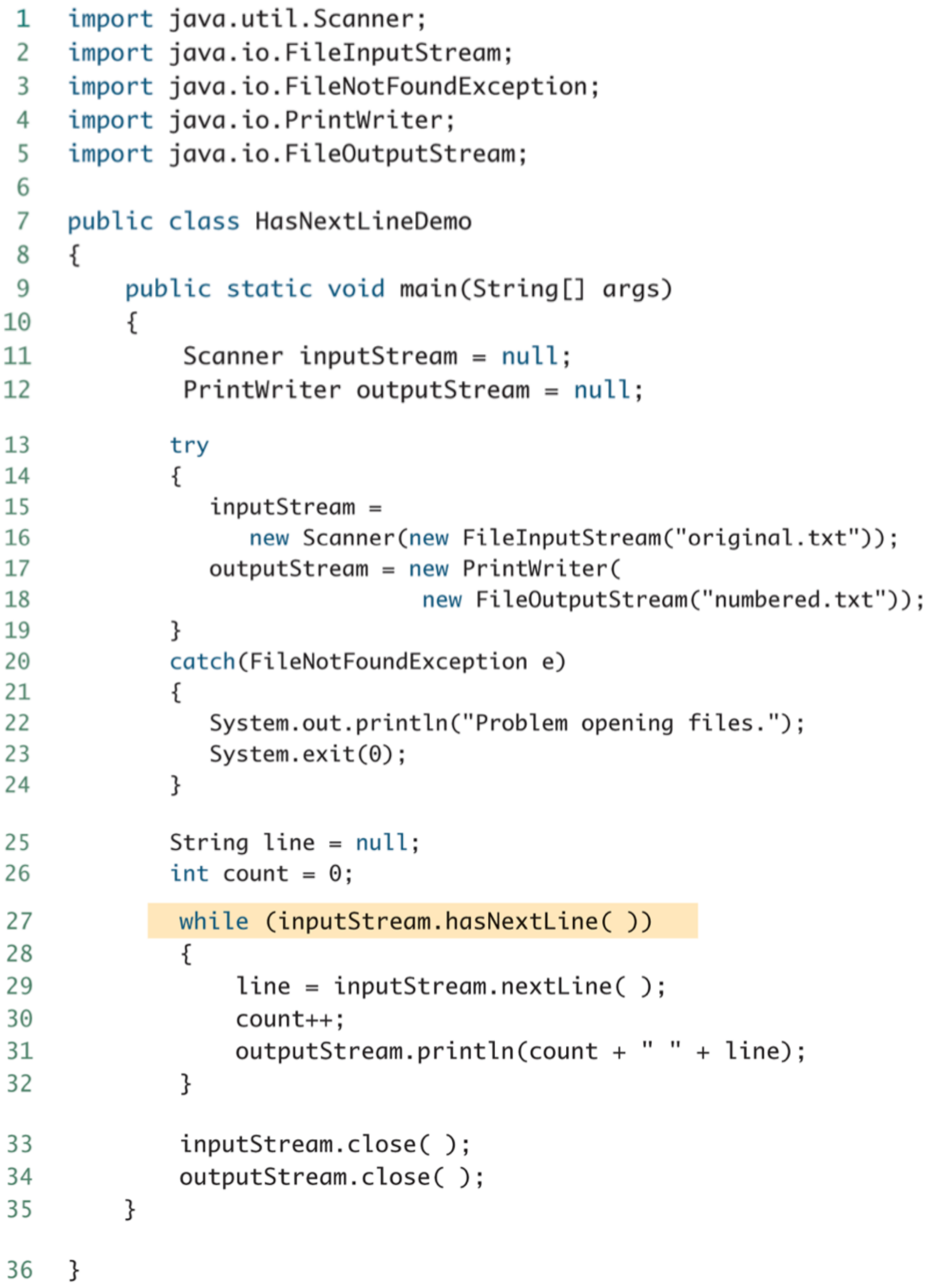
This file could have been made with a text editor or by another Java program.



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.
```

c. check end of text



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

}
```

11. Reading from a text file: BufferedReader

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

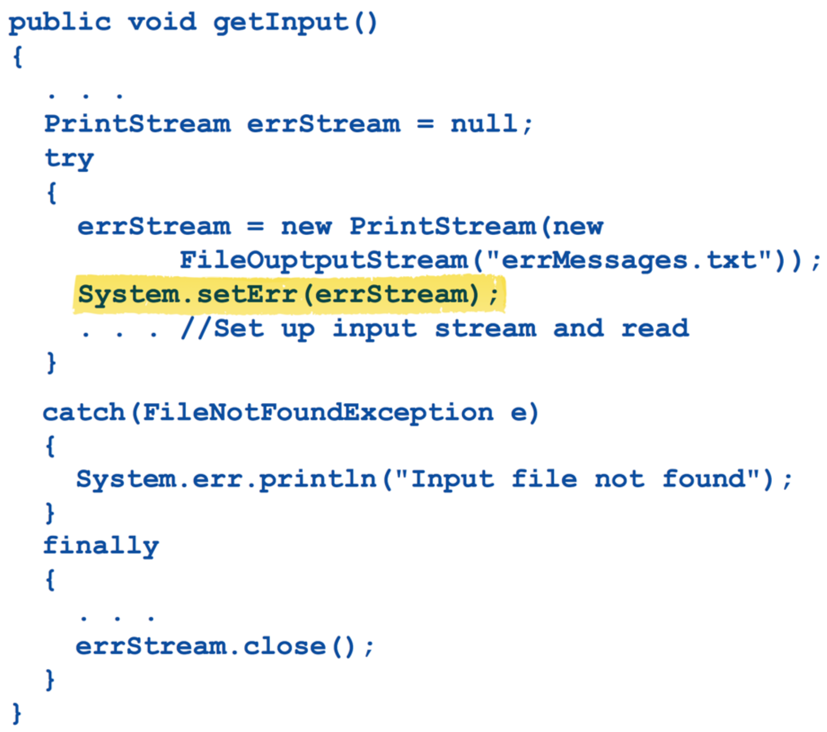
b. has `read` and `readLine` method
  - i. `read` reads a single character, and returns type `int`.
    - 1. Can use type cast:
      `char next = (char) (readerObject.read());`
    - 2. returns -1 when end of file
  - ii. `readLine` returns null when end of file

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

d. very similar to `Scanner`

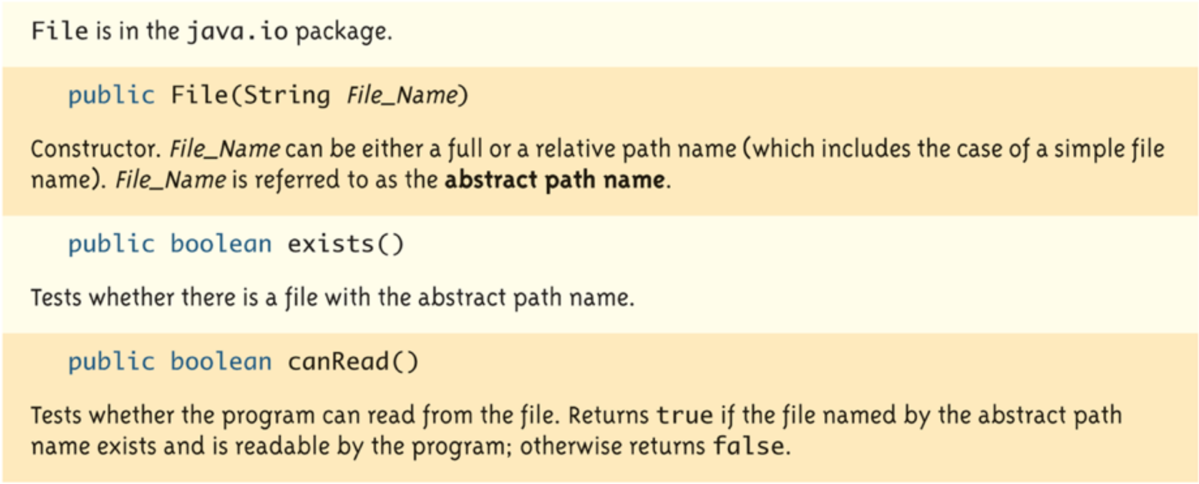
e. can’t read a number from text

12. Path Names: must be used when file not in the same directory
    - a. full path name
    - b. relative path name
13. Class `System`
    - a. `System.in`
    - b. `System.out`: normal screen output
    - c. `System.err`: error messages to the screen
    - d. redirecting standard streams:
      - i. `public static void setIn(InputStream inStream)`
      - ii. `public static void setOut(PrintStream outStream)`
      - iii. `public static void setErr(PrintStream outStream)`
      - iv. For example, instead of appearing on the screen, error messages could be redirected to a file. A new stream object should be created
      - v. Standard streams no need to be closed



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

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

14. `File` class
    a. a wrapper class for file names
    b. can be used to determined information information about the file
    c. constructor and method examples



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

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

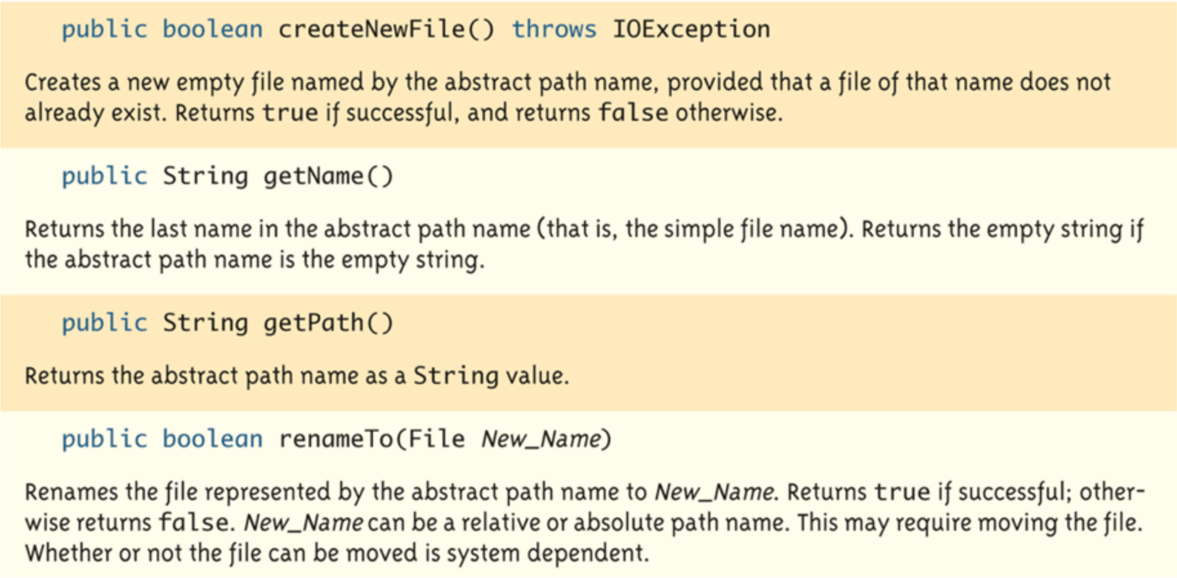
Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.

`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`.



`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.

15. Writing Simple Data to a Binary File: `ObjectOutputStream`
    a. similar to `PrintWriter` class
16. Random Access to Binary File: `RandomAccessFile`
    a. for fast access in very large databases
    b. read and write to the same file
    c. has file pointer