File Handling

Files

- Files are stored are stored on disks
- Each files consist of multiple lines composed of characters
- Each line ends with an end of line character
- The file itself may have an end of file character
- Programmers often need to read or write files stored on disks

Streams

- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

Text File I/O

- Important classes for text file **output** (to the file)
  - **`PrintWriter`**
  - **`FileOutputStream`** [or **`FileWriter`**]
- Important classes for text file **input** (from the file):
  - **`BufferedReader`**
  - **`FileReader`**
- **`FileOutputStream`** and **`FileReader`** take file names as arguments.
- **`PrintWriter`** and **`BufferedReader`** provide useful methods for easier writing and reading.
- Usually need a combination of two classes
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Output to a File

Text File Output

- To open a text file for output: connect a text file to a stream for writing

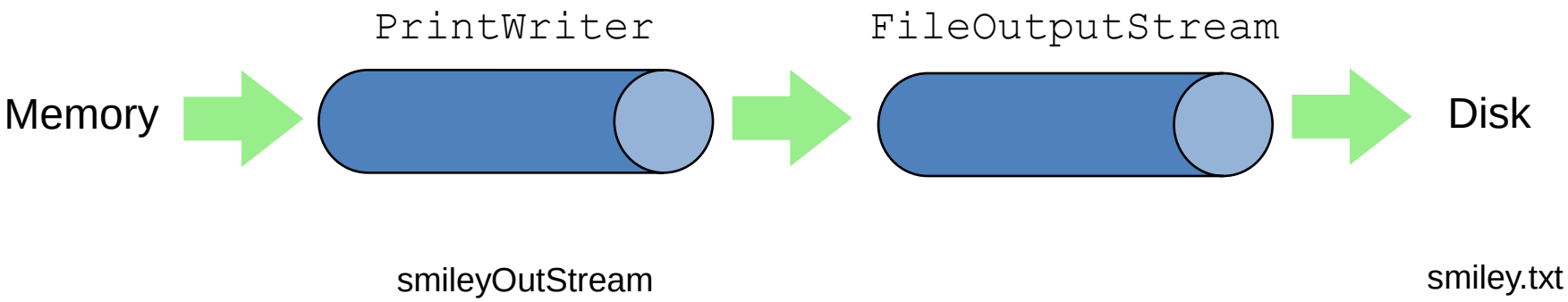
```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: create a `PrintWriter` object
  - which uses `FileOutputStream` to open a text file
- `FileOutputStream` "connects" `PrintWriter` to a text file.

Every File Has Two Names

1. the stream name used by Java
   - `outputStream` in the example
2. the name used by the operating system
   - `out.txt` in the example

Output File Streams



PrintWriter smileyOutStream = new PrintWriter(new FileOutputStream(“smiley.txt”));

Methods for `PrintWriter`

- Similar to methods for `System.out`

1. `println`

```
outputStream.println(count + " " + line);
```

2. `print`
3. `format`
4. `flush`: write buffered output to disk
5. `close`: close the `PrintWriter` stream (and file)

Example: File Output

```
public class OutputDemo{
public static void main(String[] args)
{
   PrintWriter outputStream = null;
   try
   {      outputStream =new PrintWriter(new FileOutputStream("out.txt"));
   }
   catch(FileNotFoundException e)
   {
      System.out.println("Error opening the file out.txt. " + e.getMessage());
      System.exit(0);
   }
System.out.println("Enter three lines of text:");

int count;
   for (count = 1; count <= 3; count++)
   {
   outputStream.println(count + " abc ");
   }
   outputStream.close();
   System.out.println("... written to out.txt.");
}
}
```

Overwriting/Appending a File

- Overwriting
  - Opening an output file creates an empty file
  - Opening an output file creates a new file if it does not already exist
  - Opening an output file that already exists eliminates the old file and creates a new, empty one and data in the original file is lost

```
outputStream = new PrintWriter(new FileOutputStream("out.txt"));
```

- Appending to a file
  - To add/append to a file instead of replacing it, use a different constructor for **`FileOutputStream`**:

```
outputStream = new PrintWriter(new FileOutputStream("out.txt", true));
```

  - Second parameter: append to the end of the file if it exists.

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it).
- Use the `close` method of the class `PrintWriter (or BufferedReader close` method`)`.
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

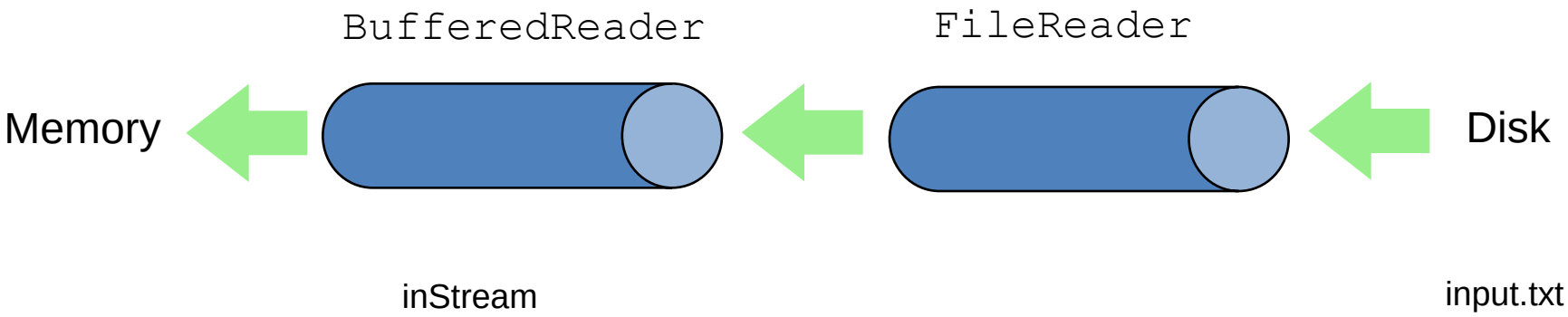
- If a program ends normally it will close any files that are open. Still the an explicit call to close should be used because :

1. To make sure it is closed if a program ends abnormally (it could get damaged if it is left open).

2. A file opened for writing must be closed before it can be opened for reading.

Input

Input File Streams




```
BufferedReader inStream = new BufferedReader( new FileReader(“input.txt”) );
```

Text File Input

- To open a text file for input: connect a text file to a stream for reading
  - a `BufferedReader` object uses `FileReader` to open a text file
  - `FileReader` "connects" `BufferedReader` to the text file
- For example:

```
FileReader s = new FileReader("input.txt");
BufferedReader inStream = new BufferedReader(s);
```

Methods for `BufferedReader`

- `read`: read a `char` at a time
- `readLine`: read a line into a `String`
- no methods to read numbers directly, so read numbers as `String`s and then convert them (`StringTokenizer` later)
- `close`: close `BufferedReader` stream

Reading Words in a String: Using `StringTokenizer` Class

- There are `BufferedReader` methods to read a line and a character, but not just a single word

- `StringTokenizer` can be used to parse a line into words
  - import `java.util.*`
  - you can specify *delimiters* (the character or characters that separate words)
    - the default delimiters are "white space" (space, tab, and newline)

Example: `StringTokenizer`

```
import java.util.StringTokenizer;


public class fileex2 {
public static void main(String[] args) {

StringTokenizer st =new StringTokenizer("This is a string");

while(st.hasMoreTokens()){

System.out.println(st.nextToken());
}
}
}
```

Testing for End of File in a Text File

- When `readLine` tries to read beyond the end of a text file it returns the special value *`null`*
  - so you can test for `null` to stop processing a text file

- `read` returns -1 when it tries to read beyond the end of a text file
  - the `int` value of all ordinary characters is nonnegative

Example: Using Null to Test for End-of-File in a Text File

```
int count = 0;
String line = inputStream.readLine();

while (line != null)
{
   count++;
   outputStream.println(count + " " + line);
   line = inputStream.readLine();
}
```

When using **`read`** test for -1

Using BufferedReader to Read from Keyboard

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class fileex3 {

public static void main(String[] args) {

BufferedReader st = new BufferedReader(new InputStreamReader(System.in));

try {
System.out.println(st.readLine());
System.out.println(st.readLine());
}
catch (IOException e) {
e.printStackTrace();
}
}
}
```

Alternative with Scanner

- Instead of `BufferedReader` with `FileReader`, then `StringTokenizer`

- Use `Scanner` with `File`:

```
Scanner inFile =new Scanner(new File("in.txt"));
```

- Similar to `Scanner` with `System.in`:

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

`File` Class `[java.io]`

- Acts like a wrapper class for file names
- A file name like "`numbers.txt`" has only `String` properties
- `File` has some very useful methods
  - `exists`: tests if a file already exists
  - `canRead`: tests if the OS will let you read a file
  - `canWrite`: tests if the OS will let you write to a file
  - `delete`: deletes the file, returns true if successful
  - `length`: returns the number of bytes in the file
  - `getName`: returns file name, excluding the preceding path
  - `getPath`: returns the path name—the full name

```
File numFile = new File("numbers.txt");
if (numFile.exists())
   System.out.println(numfile.length());
```

Reading in `int`'s

```
Scanner inFile = new Scanner(new File("in.txt"));
int number;
while (inFile.hasInt())
  {
     number = inFile.nextInt();
     // …
  }
```

Reading in lines of characters

```
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
     line = inFile.nextLine();
     // …
  }
```

`BufferedReader` vs `Scanner`

Parsing primitive types

- `Scanner`
  - `nextInt(), nextFloat(),` … for parsing types
- `BufferedReader`
  - `read(), readLine(),` … none for parsing types
  - needs `StringTokenizer` then wrapper class methods like `Integer.parseInt(token)`

Checking End of File/Stream (EOF)

- `BufferedReader`
  - `readLine()` returns `null`
  - `read()` returns -1
- `Scanner`
  - `nextLine()` throws exception
  - needs `hasNextLine()` to check first
  - `nextInt(), hasNextInt(),` …

Exercise

- Create a java program which stores rollnumber, name and marks of a student in a text file.
- It should be able to read and display these values along with the Grade of the student based on the following grading system

  0-49 → Fail

  50-59 → Pass

  60- 69 → Satisfactory

  70-79 → Good

  Above 80 → Excellent