CS 112 – MiraCosta College

Introduction to Computer Science II
Java

Module 5 – Text I/O & Streams

Chris Merrill
Computer Science Dept
cmerrill@miracosta.edu

Agenda

- Review
  - Homework, Module 3
  - Exception Handling
- Quiz on Exception Handling
- Text I/O and Streams
- Scanners and Buffered Readers
- PrintWriters
- Lab – Reading and writing text file

`try-throw-catch` Mechanism

- The basic way of handling exceptions in Java consists of the **`try-throw-catch`** trio
  - Sometimes a **`finally`** clause is added
- The **`try`** block contains the code for the basic algorithm
  - It tells Java what to do when everything goes smoothly, no problems or issues

`try-throw-catch` Mechanism

Examples of code which can throw an exception

- Opening a file for reading that doesn’t exit
- Closing a stream that is not open
- Converting a `String` to a number which is invalid
- Dividing a counting number by 0
- Accessing an array element that doesn’t exist
- Trying to access an object using a variable that is set to `null`

try-throw-catch Mechanism

- In addition, an exception can be thrown explicitly by using the **throw** statement:

```
throw new FileNotFoundException() ;
```

- The value thrown is the argument to the **throw** operator, and is always an object of some exception class. The execution of a throw statement is called *throwing an exception*

`try-throw-catch` Mechanism

- When an exception is thrown, the `catch` block begins
  - The `catch` block has one parameter
  - The exception object thrown is plugged in for the `catch` block parameter
- The execution of the `catch` block is called *catching the exception*, or *handling the exception*
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by a `catch` block

Handling Exceptions

We’ve already seen code designed to handle an exception:

```
Scanner inputFile;
try {
    File file = new File ("MyFile.txt");
    inputFile = new Scanner(file);
}
catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```

- The Java Virtual Machine searches for a catch clause that can deal with the exception.

Exception Handling with the `Scanner` Class

If a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown

- Unless this exception is caught, the program will end with an error message (i.e, it is *unchecked**)
- If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input

* Unchecked Exceptions

You have probably used methods which *could* throw an exception, but didn't appear in a try-catch block in your code

- For example, the **`nextInt`** and **`nextDouble`** methods in the **`Scanner`** class, and **`parseInt`** in the **`Integer`** class
- These exceptions are *unchecked:*
  - If an exception is thrown and not caught, then the default exception handler is invoked

Checked Exceptions

Conversely, we also have used Java methods that are *required* to be in a try-catch block

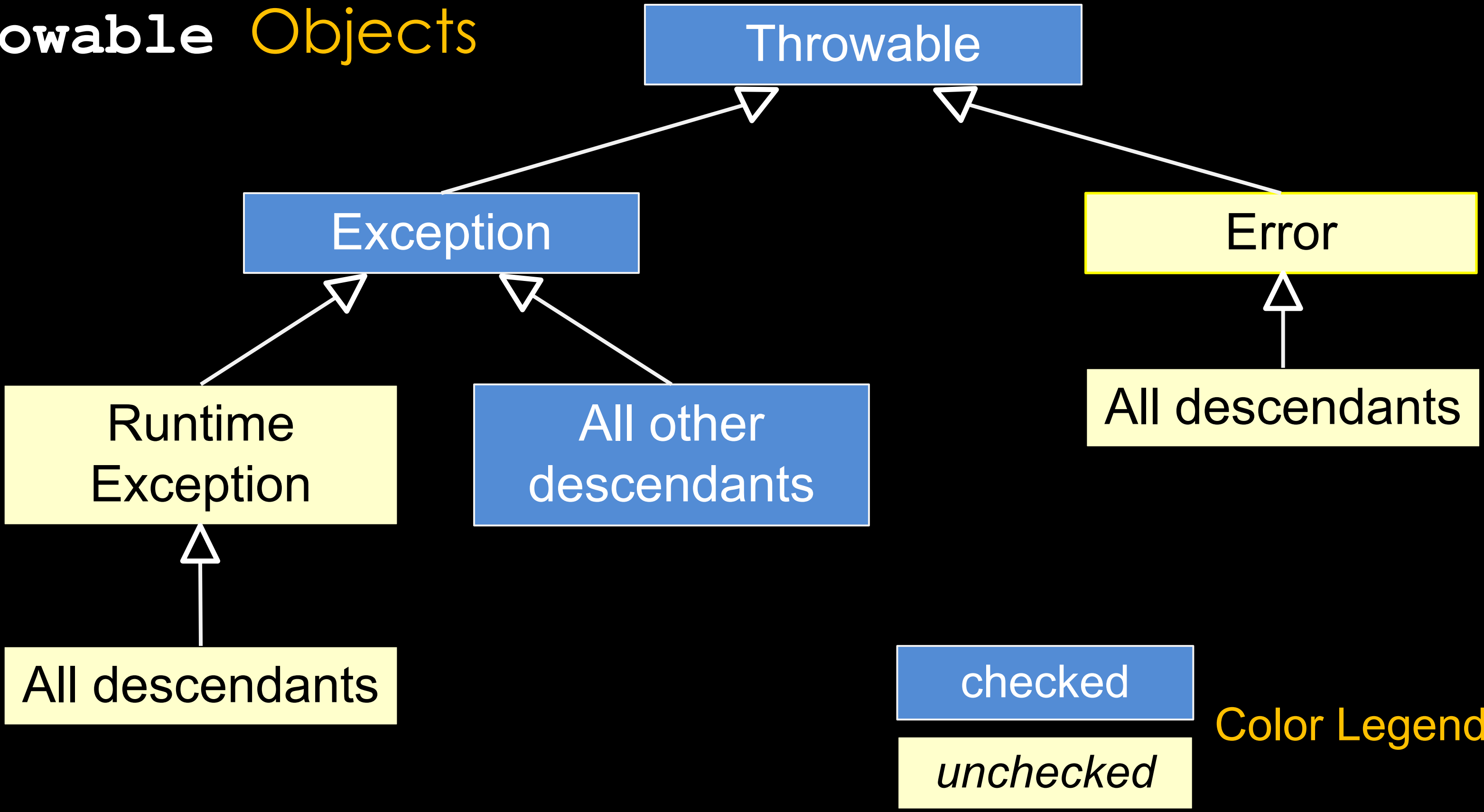
- For example, trying to open an InputStream to a file (which might not exist), or trying to close a stream (which might not be open)
- These exceptions are *checked:*
  - If an exception is *can* be thrown, then code *must* exist which handles the exception.

Exception Classes

An exception is an object.

- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the **`Throwable`** class.
- **`Error`** and **`Exception`** are derived from the **`Throwable`** class.

Hierarchy of **`Throwable`** Objects



Constructors and Accessor Methods

All exception classes (both pre-defined and programmer-defined) have the following properties:

- There is a constructor that takes a single argument of type `String`
- The class has an accessor method `getMessage` that can recover the `String` given as an argument to the constructor when the exception object was created

Using the getMessage Method

```
try {
   . . .
   throw new Exception(<StringArgument>);
   . . .
}
catch(Exception e) {
   String message = e.getMessage();
   System.out.println(message);
   System.exit(0);
}  . . .
```

Defining Your Own Exception Classes

- Exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the `catch` block
  - Different exceptions can identify different situations
- Every exception class to be defined must be a derived class of some already-defined exception class
  - These can be an exception class in the standard Java libraries, or a programmer-defined exception class

Defining Exception Classes

Constructors for exceptions are the most important members to define in an exception class

- Normally there are no other members except those inherited from the base class (i.e., the message)
- Typically two constructors are provided:

1. A no-argument constructor which provides a default exception message
2. A one-argument constructor which takes the message to be set as a string

A User-Defined Exception Class

```
public class MissingFileException extends Exception {

    // Default exception message
    public MissingFileException() {
        super("File not found! ") ;
    }

    // User-defined exception message when thrown
    public MissingFileException(String message) {
        super(message) ;
    }
}
```

Multiple **`catch`** Blocks

A **`try`** block can potentially throw any number of exception values, and they can be of differing types

- In any one execution of a **`try`** block, at most one exception can be thrown (since a throw statement ends the execution of the **`try`** block)
- However, different types of exception values can be thrown on different executions of the **`try`** block

Declaring Exceptions in a **`throws`** Clause

If a method can throw an exception but does not catch it, then the method must provide a ***`throws`*** *clause*

- The process of including an exception class in a throws clause is called *declaring the exception*

```
throws <ExceptionName>
```

- The following heading for **`aMethod`** declares that it could throw **`ExceptionName`**

```
public void aMethod() throws <ExceptionName>
```

The **`throws`** Clause in Derived Classes

- When a method in a derived class is overridden, it should have the same exception classes listed in its **`throws`** clause that it had in the base class
  - Or it should have a subset of them
- A derived class may not add any exceptions to the **`throws`** clause
  - But it can delete some

Check or Unchecked?

Finally, you must decide which exception class your class will extend, which will determine whether your exception class is checked or unchecked

- To make your class *unchecked*, it must be a descendant of the **`RuntimeException`** class
- Otherwise, it will be a *checked* exception.

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
  - The `System.in` object 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
  - The `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* if 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

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`**, **`println`**, and **`printf`**
- 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.FileNotFoundException;
import java.io.FileOutputStream;
```

The third line is optional, depending on which version of the constructor you use.

Writing to a Text File

The class `PrintWriter` has several constructors:

- One constructor takes a file name (a `String`) as its argument, either absolute or relative:

```
String myFile = "output.txt" ;
Printwriter writer = new PrintWriter(myFile) ;
```

Writing to a Text File

A stream of the class **`PrintWriter`** can be 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

Writing to a Text File

You can also use this version of the constructor to append text to the end of an existing file:

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

Writing to a Text File

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 (or use the append version on the prior slide).
- If the file does not exist, then a new, empty file named ***`FileName`*** is created
- After doing this, the methods **`print`**, **`println`**, and **`printf`** can be used to write to the file

Using a try/catch block

When a text file is opened using either method, a **`FileNotFoundException`** can be thrown

- The creation of a new **`PrintWriter`** object must be contains in a try/catch block
- If this exception is thrown means that the file could not be created.
- 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 (or the PrintWriter) connected to that file

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

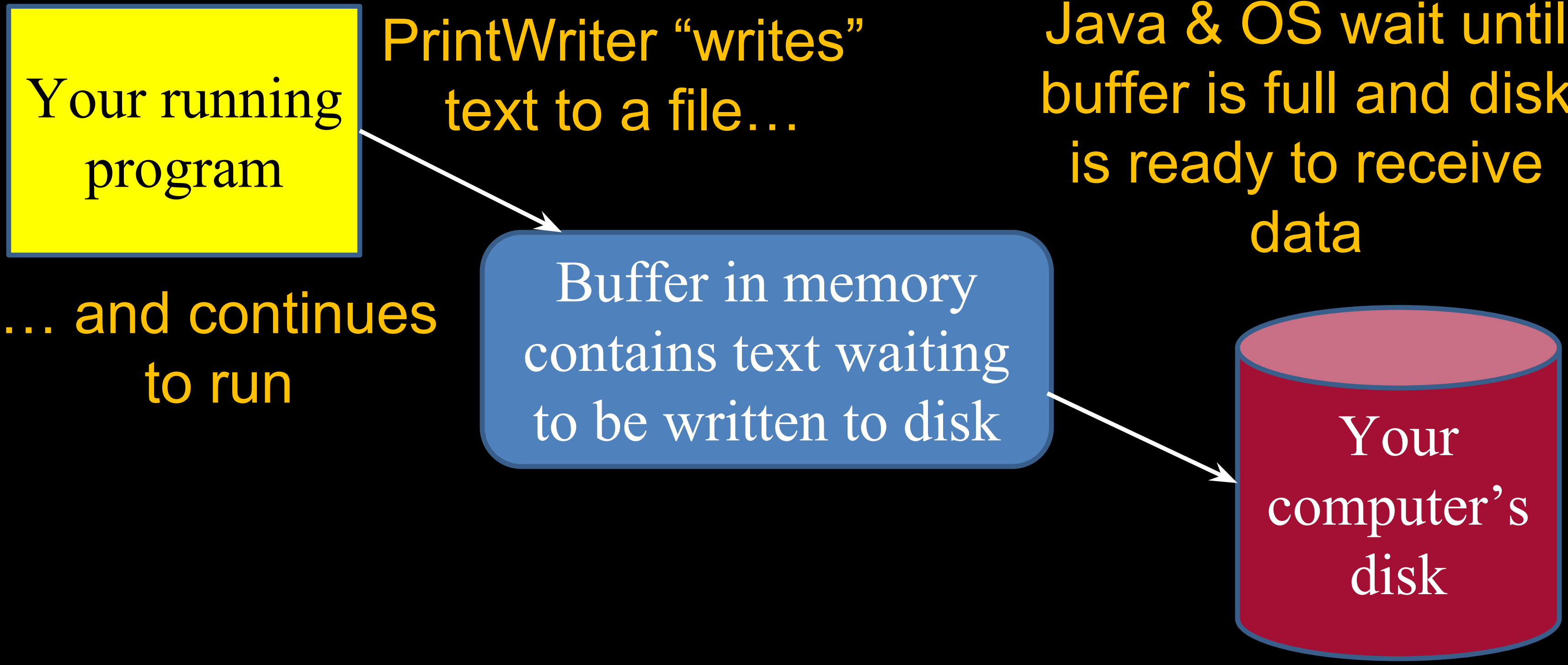
- This allows the system to release any resources used to connect the stream to the file
- If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

File Buffers and the `flush` Method

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

PrintWriter Output Buffer



Your running program

PrintWriter "writes" text to a file…

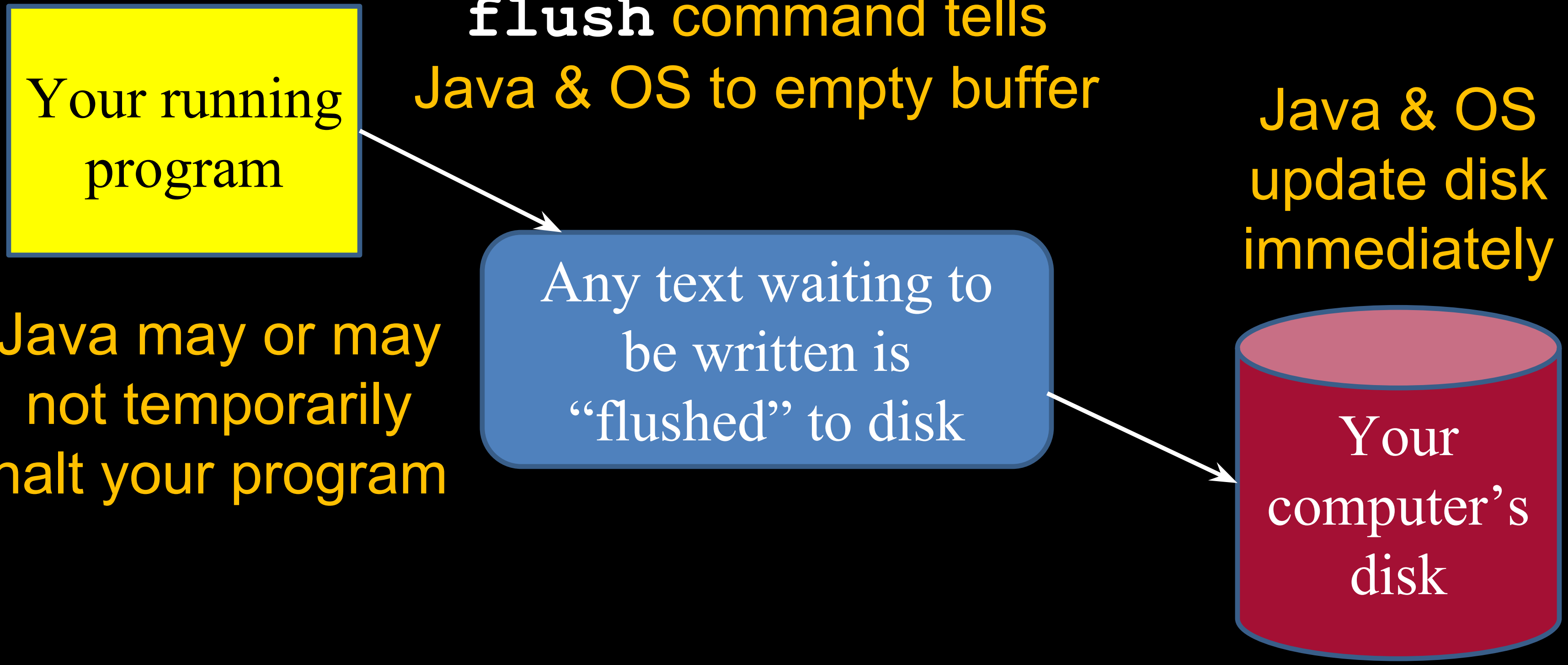
Buffer in memory contains text waiting to be written to disk

… and continues to run

Java & OS wait until buffer is full and disk is ready to receive data

Your computer's disk

`flush` empties buffer



Your running program

`flush` command tells Java & OS to empty buffer

Any text waiting to be written is “flushed” to disk

Java may or may not temporarily halt your program

Java & OS update disk immediately

Your computer’s disk

`close` invokes `flush` automatically

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

- 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

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 are unchecked:

```
NoSuchElementException
InputMismatchException
IllegalStateException
```

Unchecked exceptions are not required to be caught in a **`catch`** block or declared in a **`throws`** clause

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 **`FileOutputStream`** constructor

```
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

- We know that 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.print`**, and it will produce sensible output

- The also applies to the methods **`print`**, **`println`**, and **`printf`** of the class **`PrintWriter`**

```
writer.println(anObject);
```

Some Methods of the Class **PrintWriter**

The constructor:

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

To create a stream using a file name

```
new PrintWriter(
       new FileOutputStream(file_name)) *
```

To create a stream that appends to an existing file name

```
new PrintWriter(
  new FileOutputStream(file_name, true)) *
```

Some Methods of the Class **`PrintWriter`**

The constructor:

```
public PrintWriter(String fileName)
```

To create a stream using a file name

```
new PrintWriter("output.txt") *
```

Note: You *cannot* use this form to append to a file.

*can throw a **`FileNotFoundException`**

Some Methods of the Class **`PrintWriter`**

`public void println(`*`argument`*`)`

The argument can be a string, character, integer, floating-point number, boolean value, or an combination of these with a `+` sign. The argument can also be an object (assuming that it has a properly-defined **`toString()`** method. The line is ended with a new-line character

`public void print(argument)`

Same as **`println`**, but a new-line character is not appended to the end of the printed information, so the next output will be on the same line

Some Methods of the Class **`PrintWriter`**

`public void printf(`*`arguments`*`)`

Works the same as **`System.out.printf`**, except that output is sent to a file instead of to the screen.

`public void close()`

Closes the stream's connection to the file. The method calls **`flush`** before closing the file.

`public void flush()`

Flushes the output stream, forcing an actual physical write to the file of any data that has been buffered.

Demo Using a **`PrintWriter`**

Demonstrate the use of both constructors, and the **`print`**, **`println`**, and **`printf`** methods

Mini-Lab #1

Open a text file for output in your default folder named **`MyInfo.txt`**, then print your first name and last name (separated by a blanks) on one line. On the 2nd line, print the numbers 1 through 10 (separated by blanks).

Don't forget to close 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));
```

Using a `String` as a Parameter to the Constructor

The **`Scanner`** also has a constructor that takes a **`String`** as a parameter.

- Unfortunately, this not treated as a file name
- Instead, its treated as a **`String`** object to be scanned.

Reading From a Text File Using **`Scanner`**

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`**, **`nextDouble`**, **`next`**, and **`nextLine`** methods

“Testing” Methods in the **`Scanner`** class

- 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

Other **has**... Methods Can Validate Input

For example, the `hasNextInt()` method can be used to test if the next token in a stream is an integer:

```
total = 0 ;
while (keyboard.hasNextInt()) {
    total += keyboard.nextInt() ;
}
System.out.print("Sum is " + total) ;
```

with the input `"1 2 3 x 4"` will print `"Sum is 6"`

Methods in the Class `Scanner`

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

Constructor:

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

To read from the keyboard:

```
                new Scanner(System.in)
```

To read from a file on disk:

```
 new Scanner(new FileInputStream(filename))*
```

*can throw a `FileNotFoundException`

Methods in the Class `Scanner`

`public boolean hasNextInt()` *

returns `true` if next token is a well-formed representation of an integer

`public int nextInt()` * **

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

* throws `IllegalStateException` if stream is closed

** throws `InputMismatchException` if token is not a well-formed integer

Methods in the Class **`Scanner`**

`public boolean hasNextLong()` *

returns **`true`** if next token is a well-formed representation of an (long) integer

`public long nextLong()` * **

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

* throws **`IllegalStateException`** if stream is closed

** throws **`InputMismatchException`** if token is not a well-formed representation of a long

Methods in the Class **`Scanner`**

`public boolean hasNextShort()` *

returns **`true`** if next token is a well-formed representation of a (short) integer

`public short nextShort()` * **

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

* throws **`IllegalStateException`** if stream is closed

** throws **`InputMismatchException`** if token is not a well-formed representation of a short

Methods in the Class **`Scanner`**

`public boolean hasNextByte()` *

returns **`true`** if next token is a well-formed representation of a byte (integer number)

`public byte nextByte()` * **

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

* throws **`IllegalStateException`** if stream is closed

** throws **`InputMismatchException`** if token is not a well-formed representation of a byte

Methods in the Class **`Scanner`**

`public boolean hasNextFloat()` *

returns **`true`** if next token is a well-formed representation of a floating-point number

`public float nextFloat()` * **

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

* throws **`IllegalStateException`** if stream is closed

** throws **`InputMismatchException`** if token is not a well-formed representation of a float

Methods in the Class **Scanner**

public boolean hasNextDouble() *

returns **true** if next token is a well-formed representation of a floating-point double

public double nextDouble() * **

returns the next token as a **double**, provided the next token is a well-formed string representation of a double

* throws **IllegalStateException** if stream is closed

** throws **InputMismatchException** if token is not a well-formed representation of a double

Methods in the Class **`Scanner`**

`public boolean hasNext()` *

returns **`true`** if there is another token. May wait for the next token if using **`System.in`**.

`public String next()` * **

returns the next token

* throws **`IllegalStateException`** if stream is closed

** throws a **`NoSuchElementException`** if there are no more tokens in the stream

Methods in the Class **Scanner**

public boolean hasNextBoolean() *

returns **true** if next token is a well-formed representation of a boolean (**true** or **false**)

public boolean nextBoolean() * **

returns the next token as a **boolean**, provided the next token is either **true** or **false**

* throws **IllegalStateException** if stream is closed

** throws **InputMismatchException** if token is not a boolean

Methods in the Class **`Scanner`**

`public boolean hasNextLine()` *

returns **`true`** if there is a next line. May wait for input if the stream is **`System.in`**.

`public String nextLine()` * **

returns the rest of the current line. The terminator **`\n`** is read and discarded.

* throws **`IllegalStateException`** if stream is closed

** throws **`NoSuchelementException`** if there is no data to read

Methods in the Class **`Scanner`**

`public Scanner useDelimiter(String delims)`

Changes the delimited for input so that **`delims`** will be the only delimiter used to separate words and numbers.

You can use this the delimiter to a comma or (using a complex pattern) to any white-space character

Notice that this method returns the calling object, though it normally is used as a **`void`** *method.*

Using **hasNext**… instead of Exceptions

Demonstration of replacing a **try-catch** statement with a **hasNext**… statement

*(HasNext***Int***Demo)*

Using **`hasNextLine`** to process a file

Use the **`hasNextLine`** method to check if there is any more data to be processed in a file

(ScannerDemo)

Mini-Lab #2

Open the file you created in Mini-Lab #1 to a **`Scanner`** object. Read the first line text using **`nextLine()`** and print on the screen. Then using a **`while`** loop, read numbers from the file using **`nextInt()`** until there are no more numbers to read. Print each number on the screen as it is read.

Reading 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`**, starts with **`import`** statements:

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

Reading a Text File Using **`BufferedReader`**

Like the **`Scanner`** class, **`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 input stream that can be used as an argument to its constructor

Reading a Text File Using **BufferedReader**

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 opening the file, 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 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 is just what you think it should be)
- An invocation of **`readLine`** may throw an **`IOException`**
- Both of these exceptions *must* be handled (that is, they are *checked exceptions*)

Methods in the Class **BufferedReader**

BufferedReader is in the **java.io** package

Constructor:

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

To read from a file on disk:

new BufferedReader(new FileReader(*filename*))*

*can throw a **FileNotFoundException**

Methods of the Class **`BufferedReader`**

`public String readLine()throws IOException`

Reads a line from the input stream and returns that line. If the read goes beyond the end of the file, a **`null`** is returned.

`public int read()throws IOException`

Reads the next character in the input stream and returns the integer value of that character. If the read goes beyond the end of the file, a **`-1`** is returned.

Methods of the Class **`BufferedReader`**

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

Skips the next **`n`** characters.

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

Closes the input stream's connection to a file.

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

- 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 file

- Test for the end of the file by testing for the value **`-1`** when using **`read`**

Reading Numbers

Unlike the `Scanner` class, the `BufferedReader` class 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 convert the string into a number

Reading Numbers

- If there are multiple data items on a line, **`StringTokenizer`** can be used to decompose the original string into “tokens” (individual strings)
- If a token needs to be converted from a string to a number, then use the wrapper methods **`Integer.parseInt`**, **`Double.parseDouble`**, etc.

The **`StringTokenizer`** Class

The **`StringTokenizer`** class is used to recover the words or *tokens* in a multi-word **`String`**

- You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
- In order to use the **`StringTokenizer`** class, be sure to include the following at the start of the file:

```
import java.util.StringTokenizer ;
```

Some Methods in the **`StringTokenizer`** Class (Part 1 of 3)

- Constructors – the 1st version uses whitespace as delimiters, the 2nd explicitly defines delimiters

```
public StringTokenizer(String theString)

public StringTokenizer(String theString,
                       String delimiters)
```

- Notice that we are "overloading" the constructor for this class.

Some Methods in the **StringTokenizer** Class (Part 2 of 3)

```
public String nextToken()
public String nextToken(String delimiters)
```

Read the next “token” in the String. The 2nd version changes the delimiter string used by the 1st version.

- Both can throw **`NoSuchElementException`** if there are no more tokens to read
- Both can throw **`NullPointerException`** if **`String`** is **`null`**

Some Methods in the **`StringTokenizer`** Class (Part 3 of 3)

- Test for end of String

```
public boolean hasMoreTokens()
```

- Return the number of tokens remaining to be returned by **`nextToken()`** (i.e., using the current delimiters).

```
public int countTokens()
```

Parsing data using a `BufferedReader`

Let’s read in data from a file a line at a time, and divide it into individual data elements using a `StringTokenizer`

(BufferedReaderDemo)

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 path names are different

- 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"));
```

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*

`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

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

The System class provides three methods (setIn, setOut, and setErr) for redirecting 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();
  }
}
```

Homework

- Complete old homework and labs.
- Complete the File I/O (Stock prices) lab
- Homework, Module 5, projects 1 and 2
- Turn in everything (with Introductory Comments and documented code) at the ***beginning*** of next class

Group Lab 5

Reading a text file of stock names, symbols, and prices, each losing 1/3 of its value, and then regaining everything that was lost!