1.00 Lecture 23

Input/Output
Introduction to Streams
Exceptions

Reading for next time: Big Java 19.3-19.4

Streams

- **Java can communicate with the outside world using *streams***
- **Picture a pipe feeding data into your Java program**
  - **Where can the data come from?**
  - **Keyboard input, files, other programs, network sockets, other streams**
- **Picture a pipe leading out of your Java program**
  - **Where can the data go?**
  - **Screen output, files, other programs, network sockets, other streams**

Java I/O

- I/O -- input/output, how you get data into and out of your program
- Streams abstract away the details of I/O
  - have the same methods whatever your data source or destination
- Streams work in one direction only
  - *input streams* control data coming into program from some source
  - *output streams* control data leaving the program for some destination
  - if you want to both read and write data, you'll need two separate streams

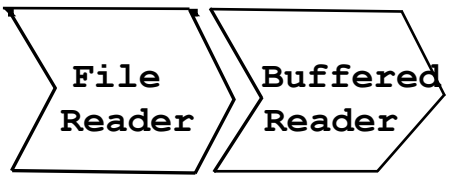
Java Stream Classes

- Java provides a hierarchy of classes for streams (in java.io.*)
  - Abstract, top-level classes that define general methods for different types of streams
    - InputStream -- reads bytes (binary data)
    - OutputStream -- writes bytes
    - Reader -- reads characters (text data)
    - Writer -- writes characters
  - Many, many subclasses implement streams
    - Some are tailored for specific data sources or destinations, e.g.,
      - FileReader reads chars from a file
    - Some add functionality to existing streams
      - BufferedReader buffers input into lines for more efficiency

System.out

- What is System.out.println()?
  - System is a special class that is automatically instantiated once when your program runs
  - It has three static member variables
    - in -- InputStream (connected to terminal input)
    - out -- PrintStream (connected to terminal output)
    - err -- PrintStream (connected to error output -- screen or special window in IDE)
  - println() is an overloaded method in PrintStream that takes a String or primitive data type as an argument, prints it to a stream and adds a line termination character.

Stream Pipeline



- We can pipeline streams to get their combined characteristics
- `FileReader` reads characters from a text file
- `BufferedReader` buffers the character stream for efficiency and allows you to read line by line (`readLine()`)

Exercise 1: Download and Run

```
import java.io.*;
public class SimpleReader {
    public static void main(String[] args) {
        try {
            String eol = System.getProperty( "line.separator" );
            FileReader fin = new FileReader("TestIn.txt");
            BufferedReader b = new BufferedReader(fin);
            FileWriter fout = new FileWriter("TestOut.txt");
            BufferedWriter bout= new BufferedWriter(fout);
            String currentLine;
            int i = 1;
            while ((currentLine = b.readLine()) != null) {
                bout.write((i++) + " " + currentLine + eol);}
            b.close();
            bout.close();
            System.out.println("Done");
        }
        catch (FileNotFoundException ef) {   // Later in lecture
            System.out.println("File not found");}
        catch (IOException ei) {
            System.out.println("IO Exception"); }
} }
```

Exercise 2

- Create new text file of 10 lines for your program to read
  - File-> New -> File, name it 'TestIn1.txt'
  - Modify SimpleReader to read the new file and write TestOut1.txt
  - After the program runs, hit Refresh (F5) in Eclipse Explorer to see the file
  - Try to read a file that doesn't exist.
- Change the while statement to (a bad idea):

```
while (b.readLine() != null) {
   fout.write((i++) + " " + b.readLine() + "\n");}
```

  - What happens, and why? (It's a common error)
- "Accidentally" write to your input text file (e.g. TestIn.txt)
  - Make a copy of your input text file first. What happens?
- Other notes:
  - Always check for end of file (EOF):
    - readLine() returns null
  - Always close your streams when done: saves system resources, avoids file corruption if system crashes

The 3 Flavors of Streams

In Java, you can read and write data to a file:

- as text using `FileReader` and `FileWriter`
- as binary data using `DataInputStream` connected to a `FileInputStream` and as a `DataOutputStream` connected to a `FileOutputStream`
- as objects using an `ObjectInputStream` connected to a `FileInputStream` and as an `ObjectOutputStream` connected to a `FileOutputStream`

Parsing

- readLine() is ok if you want to read whole lines
- read() is ok if you want to read character by character
- What if you have structured data?
  - meaning is dependent on position or formatting
  - comma-separated values (or other delimiters/separators)
- Reading this data in a meaningful way is called *parsing*

Parsing

- When you parse (tokenize) a file, you are looking for *tokens*
  - Sequences of one or more characters that "belong" together
  - Sometimes tokens are separated by *delimiters* (",", "\t", " ", "\n"), sometimes not
- Three ways to parse in Java
  - Use the `split()` method in `String` class with regular expressions. Simplest way to parse simple delimited files.
  - `StreamTokenizer` : works with `Stream`, reads token by token, treats delimiters as tokens, recognizes "words" and "numbers"
  - `Scanner` : default use is simple, works with files and `String`s

`String`'s `split()` Method

- Call `split()` on the `String` you want to parse and get back an array of `String`s parsed into "tokens".
- Argument is the delimiter you want to use.
- Characters `[]\^$.|?*+()` have special meaning in `split()` and must be ***escaped*** to use them as a delimiter, e.g., `"\."` to use a period as a delimiter.
- The `String` "`\\s`" as an argument to `split()` means use any whitespace (`" "`,`"\t"`,`"\n"`...) as a delimiter.
  - "`\\s+`" means use any succession of whitespace characters as a (single) delimiter.

```
String s = "James Bond,3-0007,10-250";
String[] parts = s.split(",");
// parts = { "James Bond", "3-0007", "10-250"};
```

Exercise 3

- Write a new class called `WordCount`, based on `SimpleReader` above.
- Read `TestIn` just as before, but instead of writing it back out, use `split()` to count the number of "words" on each line.
- Sum them as you read the file, and output the word count at the end.

Errors

- Error is any condition that produces unwanted or incorrect results. Strategies to deal with errors:
  - Anticipate where errors might occur, and program to prevent them if possible
    - We do this 99% of the time
  - Catch errors as they occur; possibly reroute execution flow, set alternate values, show messages
    - You may not anticipate everything that can go wrong
    - Some errors may be out of your control (e.g. input)
    - You still need to handle these, however
  - Java exceptions are how we handle this second group of errors

Exceptions

- Exceptions are how Java handles errors that the method in which the error occurs can't handle
  - Exceptions are objects that are thrown (created and sent to calling methods) in response to runtime errors
  - Runtime errors can occur from attempting illegal operations, invalid input, corrupted or unavailable resources, system problems, …

Exceptions: Try, throw, catch

- The Java exception mechanism has three elements:
  - Throw (what a method does)
    - If method detects error that it cannot handle,
    - Method throws an exception
      - Method returns either its usual return value or an exception object
  - Try block (what the caller of the method does first)
    - Called methods that may throw an exception are placed in a try block (defined by curly braces) in the calling method
      - A regular return value continues program flow as usual.
      - An exception return causes execution to go to the catch block.
  - Catch blocks follow try blocks (what caller does on error)
    - Each block contains an exception handler of a given type
    - Exception objects have types; different types can be handled by different catch blocks, each with logic specific to the exception

Catching an exception

```
import javax.swing.*;

public class BadInput {
   public static void main(String[] args) {
     while (true) {
        String answer = JOptionPane.showInputDialog("Enter an
            integer (0 to quit)");
        int intAnswer = -1;    // Must declare outside try block
        try {                                    // Try block
            intAnswer = Integer.parseInt(answer); // Throw
             System.out.println(intAnswer);       // Regular flow
        } catch (NumberFormatException e) {       // Catch block
            JOptionPane.showMessageDialog(null, "Not an integer");
        }
        if (intAnswer == 0)
            break;
     }
     System.exit(0);
   }
}
```

Exercise 4

- **Download BadInput from the Web site**
- **Comment out:**
  - **Try block ('try' and the curly braces; leave intAnswer = …),**
  - **Catch block (remove the entire block, including code)**
  - **Save/compile**
- **Enter non-integer input. See what happens.**
  - **What happens if the user types a non-integer, "Cathy", for example?**
  - **Is this what we've been doing so far in 1.00 for input?**
- **Then remove the comments, restoring the try/catch blocks**
  - **Save/compile**
  - **Enter non-integer input.**
  - **What happens?**

Exceptions, Streams, Inheritance

- Since exceptions are objects, exception classes may use inheritance. `FileNotFoundException` is a subclass of `IOException`.
- When an error is detected, Java will create and throw a new instance of an appropriate type of exception.
- The first catch statement matching the exception class or one of its superclasses is executed.
  - The order of the catch blocks matters

Exception Inheritance Example

```
try                    // From exercise 1
{
  // Read file. If bad file, throw exception
  // If file ok, continue execution as usual
  FileReader fin = new FileReader( "TestIn.txt" );
  // Other statements follow, but need value of "fin"
  // They can't be executed if the line above didn't work
}
catch ( FileNotFoundException ef )
{
  // Handle not finding the file (bad file name, no permission…)
}
catch ( IOException ei )
{
  // Handle any other read error (disk crashed…)
}

// If we reversed these catch blocks, the program
// would not compile (unreachable code)
```

Checked vs. Unchecked Exceptions

- Java distinguishes between *checked* and *unchecked* exceptions.
- Checked exceptions are those which the programmer must handle at runtime, such as a `FileNotFoundException`.
  - These are generated by user, not programmer, error.
  - Example: All `IOException`s are checked exceptions.
  - If you call a method that can throw a checked exception, you must put the method call in a try block and catch the exception.
- You do not have to handle unchecked exceptions
  - Unchecked exceptions are the result of programmer error, so the best way of handling them is to fix the program
  - Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`.

When to Use Exceptions

- Why do we need exceptions?
  - Usually errors are caught in a low-level routine: file reader or math function that is very general-purpose and has no idea whether the error is serious or not
  - The caller (user) of that routine is the one who knows the context of the error and can decide the best course of action. After an error in an I/O reader:
    - A mangled Tweet message can be ignored
    - A mangled turn left message to an aircraft cannot be ignored
- If you can fix an error locally, don't use an exception
  - Exceptions are used when the method can't fix the error itself
- In 1.00, your primary use of exceptions will be in stream I/O and sensor I/O, where their use is required

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012