MIT AITI
Lecture 15: I/O and Parsing

Kenya 2005

What we will learn in this Lecture.

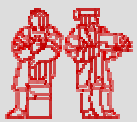
- This Lecture is divided into 2 main parts:

I – Input /Output:

- Input vs Output, and Byte vs Character Streams
- Important Stream Classes and Using these Classes
- Example of Reading from and Writing to Text Files
- Example of Reading text from Keyboard input
- Using buffered streams

II – Introduction to Parsing:

- Delimiters
- **`StringTokenizer`**



I/O Basics

- I/O = Input/Output – Communication between a computer program and external sources or destinations of information
- Involves: - Reading input from a source
  - Writing output to a destination
- Reading and Writing is specified by 4 abstract classes:

- `Reader`
- `Writer`
- `InputStream`
- `OutputStream`

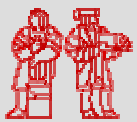
Java I/O Streams

- Java programs communicate with the outside world using *Streams*

- *Streams* are used for reading and writing data

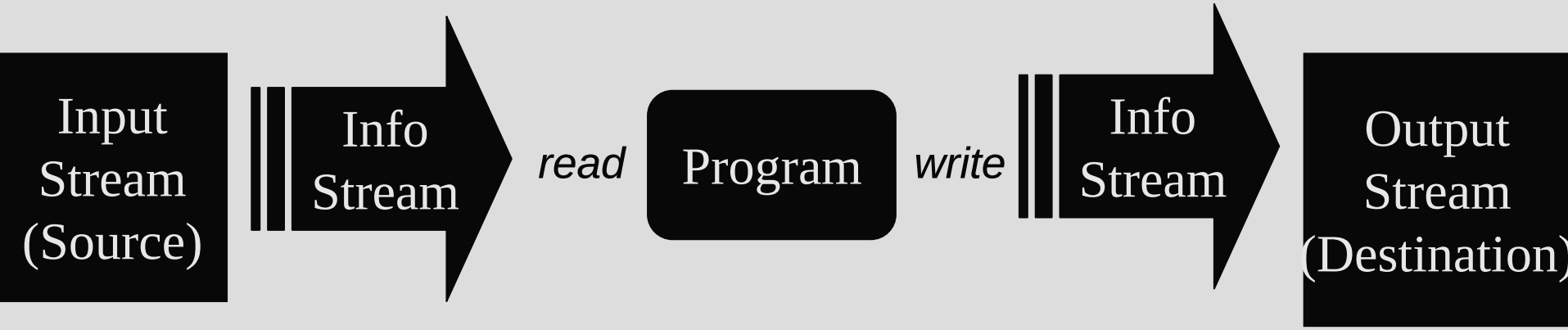
- I/O Streams are unidirectional
  - Input stream for data coming into program
  - Output stream for data leaving program

- Examples of Sources and Destinations of info include: Files, Network connections, other programs, etc.

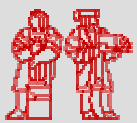


Input vs Output Streams

- An object from which we can read data is an *Input Stream*



- An object to which we can write data is an *Output Stream*



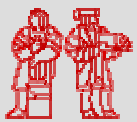
Byte vs. Character Streams

- **Byte Streams** are used to read and write data which is in binary format (1's and 0's)

  e.g. images, sounds, etc.

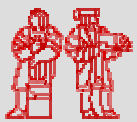
- **Character Streams** are used to read and write data which is in text format (characters)

  e.g. plain text files, web pages, user keyboard input, etc.



Important Stream Classes

- **FileInputStream**
  - *Read data in binary format from files*
- **FileOutputStream**
  - *Write data in binary format to files*
- **FileReader**
  - *Read text data from files*
- **FileWriter**
  - *Write text data to files*

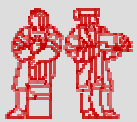


Using a Stream class

1. Open a stream by instantiating a new stream object

2. While more information to read/write, read/write that data using methods in the Stream Classes

3. Close the stream by calling the object's `close()` method

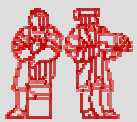


Java I/O Classes

- The `java.io` package offers classes used to read/write data from/to files

- To read/write data, we instantiate a subclass of one of the 4 abstract superclasses:

	input	output
byte	`InputStream`	`OutputStream`
character	`Reader`	`Writer`

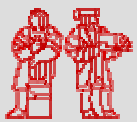


Using `Reader`

- Recall: a `Reader` is used to read a character input stream
- `Reader` offers methods to read single characters and arrays of characters. E.g.

```
int read()
```

- `Reader` is abstract so you **must** instantiate a **subclass** of it to use these methods



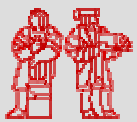
Reading from a Text File

```
public void readFile() {
   FileReader fileReader = null;
   try {
      fileReader = new FileReader("input.txt");
     int c = fileReader.read();
      while (c != -1) {
         char d = ((char)c);
         c = fileReader.read();
      }
   } catch (FileNotFoundException e) {
      System.out.println("File was not found");
   } catch (IOException e) {
      System.out.println("Error reading from file");
   }
   if (fileReader != null) {
      try { fileReader.close(); }
      catch (IOException e) { /* ignore */ }
   }
}
```

Step 1 → `fileReader = new FileReader("input.txt");`

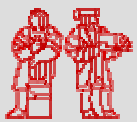
Step 2 → `while (c != -1) { char d = ((char)c); c = fileReader.read(); }`

Step 3 → `try { fileReader.close(); } catch (IOException e) { /* ignore */ }`



BufferedReader

- `BufferedReader` is a subclass of `Reader`
- Buffers the character stream from `FileReader` and has `readLine()` method to read an entire line of characters efficiently
- ```
  FileReader fr = new FileReader("myFile.txt");
  BufferedReader br = new BufferedReader(fr);
  ```
- The `readLine()` method returns `null` when there are no more lines to read

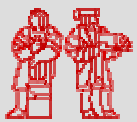

  ```

Using BufferedReader

```
public void readFileWithBufferedReader() {
    BufferedReader bufferedReader = null;
    try {
        FileReader fr = new FileReader("input.txt");
        bufferedReader = new BufferedReader(fr);
      String line = bufferedReader.readLine();
        while (line != null) {
            // do something with line
            line = bufferedReader.readLine();
        }
    } catch (FileNotFoundException e) {
        System.out.println("File was not found");
    } catch (IOException e) {
        System.out.println("Error reading from file");
    }
    if (bufferedReader != null) {
        try { bufferedReader.close(); }
        catch (IOException e) { /* ignore */ }
    }
}
```

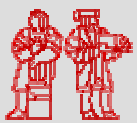
POP QUIZ

- Why can we not create instances of the `Reader` class directly?
  `Reader` is an Abstract class, and cannot be instantiated
- Which kind of stream would we use to read/write data in binary format?
  Byte Streams
- Which kind of stream would we use to read/write data in text format?
  Character Streams
- Why do we wrap a `FileReader` with a `BufferedReader` before reading from a Text file?
  BufferedReader has the readLine() method used to read entire lines



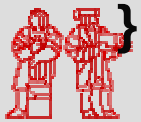
Writer

- `Writer` is an abstract class used to write to character streams
- Offers `write` methods to write single characters, arrays of characters, and strings (look at API)
  `e.g. void write(int c)`
- `BufferedWriter` (subclass of `Writer`) offers efficient writing; `newLine()` method to insert a blank line and `write(String n)` method to write data
- Close `Writer` with `close()` method when done



Writing to a Text File

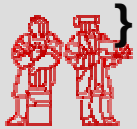
```
public void writeFileWithBufferedWriter() {
    BufferedWriter buffWriter = null;
    try {
        FileWriter fw = new FileWriter("output.txt");
        buffWriter = new BufferedWriter(fw);
        while (/*still stuff to write */) {
            String line = // get line to write
            buffWriter.write(line);
            buffWriter.newLine();
        }
    } catch (IOException e) {
        System.out.println("Error writing to file");
    }
    if (buffWriter != null) {
        try { buffWriter.close(); }
        catch(IOException e) { /* ignore */ }
    }
}
```



Example: Copying Text Files

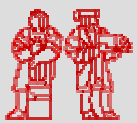
```
void copyFiles(String inFilename, String outFilename)
    throws FileNotFoundException {
  BufferedReader br = null;
  BufferedWriter bw = null;
  try {
    br = new BufferedReader(new FileReader(inFilename));
    bw = new BufferedWriter(new FileWriter(outFilename));
    String line = br.readLine();
    while(line != null) {
      bw.write(line);
      bw.newLine();
      line = br.readLine();
    }
  } catch (IOException e) {
    System.out.println("Error copying files");
  }

  if (br != null) {try {br.close();} catch(IOException e) {}}
  if (bw != null) {try {bw.close();} catch(IOException e) {}}
}
```



Reading From Keyboard Input

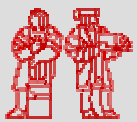
- Keyboard input is sent over a `Stream` referred to as "standard" input, but to read the data you want it to be a `Reader`
- `InputStream` acts as a crossover class, to get from a `Stream` to a `Reader`
- To read characters over an `InputStream`, need to wrap it in an `InputStreamReader`
- To read line by line, wrap the `InputStreamReader` with a `BufferedReader`



Example: Reading from Keyboard Input

```
/**
 * Returns a line read from keyboard input.
 * Return null if there was an error reading the line.
 */
public void String readKeyboardLine() throws IOException {
   BufferedReader br = null;
   String line = null;
   try {
      br = new BufferedReader(new InputStreamReader(System.in));
      line = br.readLine();
   } catch (IOException e) {}

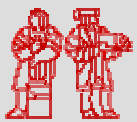
   if (br != null) {
      try { br.close(); }
      catch (IOException e) { /* ignore */ }
   }
   return line;
```



Streams Conclusion

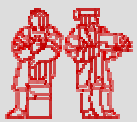
- Make sure you look at the `InputStream` and `OutputStream` hierarchy, and `Reader` and `Writer` hierarchy in a Java Textbook to see their subclasses and methods

- Use Java API!!!



Introduction to Parsing

- Programs often encode data in text format before it is stored in files
- Programs later need to decode the text in the files back into the original data
- Process of decoding text back into data is known as *parsing*

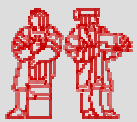


Delimiters

- When data is stored in text format, *delimiter* characters are used to separate *tokens* (or pieces) of the data
- A list of first names stored separated by the '#' delimiter: `Greg#Kwame#Sonya#Bobby`
- Same list with a newline delimiter:

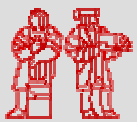
```
Greg
Kwame
Sonya
```

- Other common delimiters are '|' ':'



StringTokenizer I

- When trying to read a line of input, we get one long string.
- We need to find the *delimiters* in the long string and separate out each of the individual pieces of information (tokens)
- For this, we use the `StringTokenizer` class in java.util



StringTokenizer I

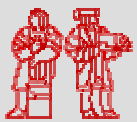
- When constructing the tokenizer object, you can specify which characters are the delimiters in your case

- Default constructor will assume " \t\n\r" to be delimiters

```
StringTokenizer r = new StringTokenizer(line);
```

- Second constructor accepts `String` of any delimiter characters

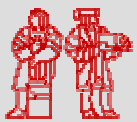
```
String line = myFile.readline();
StringTokenizer t = new StringTokenizer(line, "#");
StringTokenizer s = new StringTokenizer(line, ",\&\|");
```



StringTokenizer II

- Useful `StringTokenizer` methods:

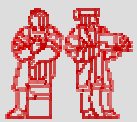
- `String nextToken()` method returns the next data token between delimiters in the text
- `boolean hasMoreTokens()` returns true if the text has remaining tokens



Using StringTokenizer

- Printing out every name from a file where names are delimited by whitespace:

```
public void printNamesFromFile(String filename) {
  BufferedReader br = null;
  try {
    br = new BufferedReader(new FileReader(filename));
    String line = br.readLine();
    while(line != null) {
        StringTokenizer st = new StringTokenizer(line);
        while(st.hasMoreTokens()) {
          System.out.println(st.nextToken());
        }
        line = br.readLine();
    }
  } catch (IOException e) {
    System.out.println("Error reading from file.");
  }
  if (br != null) { try { br.close(); } catch(IOException e) {} }
}
```

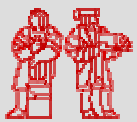


Parsing Numbers

- Often necessary to parse numbers stored as text into Java primitives
- Wrapper classes for primitives provide static methods to do so

```
int Integer.parseInt(String s)
double Double.parseDouble(String s)
```

- Throw `NumberFormatException` if the specified `String` cannot be converted into the primitive



EC.S01 Internet Technology in Local and Global Communities
Spring 2005-Summer 2005