Féidearthachtaí as Cuimse
Infinite Possibilities

File Processing

Object Oriented programming

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TECHNOLOGICAL UNIVERSITY DUBLIN

Bits, bytes, characters

- A bit stores a 1 or 0
- Anything with two separate states
- The smallest building block of storage

- A **byte** - a grouping of 8 bits
- e.g. 01100110
- How many unique patterns can a byte have?

Characters

- A byte has 256 different possible values

- A byte is enough to store one “character” (i.e. something meaningful to humans);
  - e.g. N 2 @ ũ æ Ω ओ 김 И are all chars

- Common character encoding sets:
  - ASCII (each character is 7 bits)
    - e.g. A = number 65 in ASCII or 01000001
  - UTF-8 (i/net standard) = 1- 4 bytes
    - First 8 bits = backward compatability with ASCII

Files

- Persistent data
- Binary file – not human readable – requires a specific computer program (e.g. database files)
- Text file – human readable – *usually* ASCII

- *Our focus: sequential (as opposed to random) access text (as opposed to binary) files,*

What use are text files?

- In application, persistent data usually involves a database (e.g. facebook, DIT system, banking systems etc)

- But text files still useful – quick access for specific info; text data (e.g. emails, posts etc), word processing files etc;

Using a file - Typically any of:

- **Open** an existing file
- **Read it** – line by line
- **Write out** data – line by line
- **Create a** new file
- **Close**

Opening

- Open (create a handle to it)

```
File fleExample = new File("Example.txt");
```

- What might go wrong? (causing an "Exception")

***** In Eclipse, the default file location is the project directory - not the package or src directory. This will save you a lot of aggro! ******

Reading a file

Several ways in java

(1) Java.util.Scanner class

```
File fleExample = new File("Example.xpl");
Scanner myScanner = new Scanner(fleExample);
```

Scanner class: breaks input into tokens using a “delimiter” which is whitespace by default

(An aside: what can you tell from above about the constructor of the Scanner class?)

“Scanner” class methods

Method name	Description
next()	reads and returns the next token as a String
nextLine()	reads and returns as a String all the characters up to the next new line (\n)
nextInt()	reads and returns the next token as an int, if possible
nextDouble()	reads and returns the next token as double, if possible
hasNext()	returns true if there is still a token in the Scanner
hasNextLine()	returns true if there is still at least one line left to be read in the Scanner
hasNextInt()	returns true if the next token can be read as an int
hasNextDouble()	returns true if the next token can be read as an double

Scannner Recap

- A Scanner is a **text parser** — it reads a sequence of characters from some source (file, keyboard, network, etc.) and breaks it into **tokens** (words, numbers, lines).

- But Scanner itself **does not know** where the characters come from — it just knows *how to interpret them*.
  That's why it needs a **stream or source** to feed it data.

For example

- If a file has this in it

```
Mary O'Neill

John Waters

Frederick O'Meara


...
```

Or this

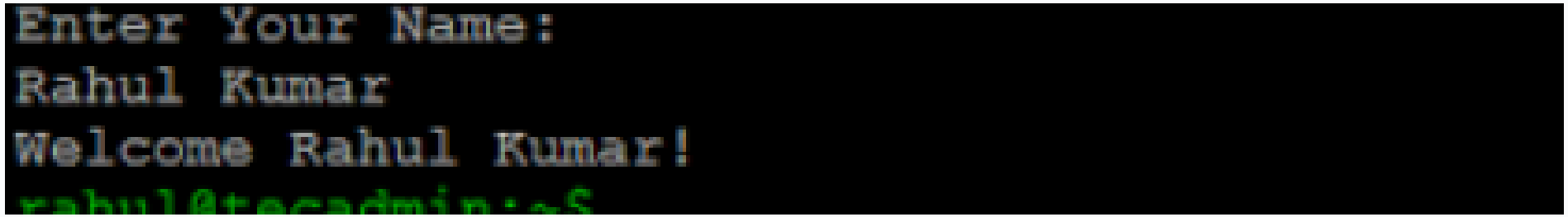
```
Mary O’Neill, 12 Oldbawn Road
John Waters, 45 Avondale Gate
Frederick O’Meara, 25 Cooper Lane

…
```

```
scanner.useDelimiter(",");
```

Scanner class for user input

- **Separate NOTE**
- can also be used **for input at the console**




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

String text = input.nextLine();
```

Reading through the file contents with Scanner class

```
   // while there are lines in the file…
 while (myScanner.hasNextLine())
   {
   // get the next line.. And do whatever it is you want with
it..
```

and close it

```
    }
myScanner.close();
```

Writing to a file

- PrintWriter class is useful

```
// first, get a "print writer" object
      PrintWriter myOutFile =new PrintWriter(someFile);

      // then, use a method from the printwriter class
      // to write to the file  e.g. println()
         myOutFile.println("whatever you want to write")
     }
```

Writing to a file

- Close the printwriter

```
myOutFile.close();
```

Formatting

- **Fixed-width Formatting (for reports or columns)**
- **Add Headers and Footers**
  - Give structure to text files (useful in reports).
  - out.println("===== Library Report =====");
  - out.println("Generated on: " + LocalDate.now());
  - out.println("-------------------------");
  - // ... data lines ...
  - out.println("============================");

Formatting Text

- String formatted = String.*format*("%-4d | %s%n", count++, line);
- myOutFile.println(formatted);

Append

Why we use FileWriter with PrintWriter for append mode

- PrintWriter by itself doesn't know *how* to open or append to a physical file.
  It just knows **how to print text** to whatever output stream you give it.
- FileWriter, on the other hand, **controls the file connection** —
  and that's where you can specify *append mode*.

```
public static void appendStudent(Student s, String fileName) {
    try {
        // 1. Create a FileWriter in append mode (true)
        FileWriter fw = new FileWriter(fileName, true);
        PrintWriter myOutFile = new PrintWriter(fw);

        // 2. Write student details as a CSV line
        myOutFile.println(s.getName() + "," + s.getAge() + "," + s.getCourse());

        // 3. Close the writer
        myOutFile.close();

        System.out.println("Student appended successfully.");

    } catch (FileNotFoundException e) {
        System.out.println("Error appending to file: " + e.getMessage());
    }
}
```

A note on class structure

- If you have a java application that uses file from multiple classes
- Usually better to put the **File functionality** into its own class

Things can go wrong at runtime

- TRY it out

and

- CATCH the (right) error

Demo

What can go wrong?

When you work with **file input/output** (reading or writing files), several things can go wrong:

- The file might not exist.
- The program might not have permission to access it.
- The disk could be full or unavailable.
- The file path might be invalid.

Exceptions

- Exceptions are java's runtime management technique for that run time errors can be handled with grace. Exceptions represent errors or unusual conditions that occur while a program is running (e.g. file not found, invalid input). Instead of crashing the program immediately, Java lets you signal and handle these problems using exceptions, making code more robust and easier to debug.

Try/ Catch

```
try     {
  PrintWriter myOutFile =new     PrintWriter(someFile);
   myOutFile.println("whatever you want to write")
   myOutFile.close();
}
catch (FileNotFoundException e)   // "Exception" is the most general one, but better to catch
                                                    the specific error that may happen
{   You could also use IOExecpetion
  System.out.println("error occurred" + e.getMessage())
}
```

How to know what error to catch?

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source` - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

Scanner

- If you are using API classes - the API lists the error(s) that might be thrown
- If you are writing custom methods, you define what exceptions may happens

Exceptions in java

- Are used so that run time problems can be handled gracefully, instead of crashing
  - Better User experience
    - E.g. using a java ATIM application where file is not found
      - Runtime crash of java code and technical error (i.e. unhandled exception)

    Versus

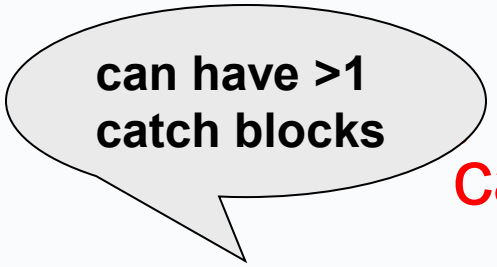
      - “We’re sorry. An error has occurred. Please try again later”

Try/ Catch for exception handling

- to detect and handle exceptions use

```
try {
        …
catch (SomeException e1) {
        …
  // e.g. System.out.println("Error: +
"e.getMessage());

}
finally
{ // Always executes. Optional to use
        …
}
```



Typically, Catch the error but

- You can instead throw the error on to whatever is running it instead of using try/catch

Throws the exception on (instead of try/ catch)

```
public void readFile() throws FileNotFoundException
{

 File fleExample = new File(fileName);
 Scanner myScanner = new Scanner(fleExample);

  while (myScanner.hasNextLine())
    {
      String line = myScanner.nextLine();
      System.out.println ("I read this line " + line);
    }

  myScanner.close();

} // readFile
```

Exceptions Summary

- To handle a message such as FileNotFoundException, you need to implement either
  - a try (for the risky code), followed by a catch of the exception
  - OR throw the error on in the method header where the risky code is being called method name ( ) throw Exception.

```
try {

FileReader fr = new FileReader("data.txt"); // use fr ...

}

catch (FileNotFoundException e) { System.out.println("File not found: " +
e.getMessage());

}
```

Covered :

- Types of files
- Common file operations – open, read, write, close etc
- Common file classes and methods (File, Scanner, PrintWriter etc.) in the API
- Scanner methods for navigating
- Try / Catch