

# COMPSCI4039: Programming

FileIO and Exceptions

# What we're working towards

- Loading a comma separated file holding student names and grades into an array of Student objects
- The file parsing bit is quite straightforward
- But the **exception handling** is a bit of a pain.

# Exceptions

- How many of you have seen an Exception?

All of you...

```
Exception in thread "main" java.util.IllegalFormatConversionException: f != java.lang.Integer
    at java.util.Formatter$FormatSpecifier.failConversion(Formatter.java:4302)
    at java.util.Formatter$FormatSpecifier.printFloat(Formatter.java:2806)
    at java.util.Formatter$FormatSpecifier.print(Formatter.java:2753)
    at java.util.Formatter.format(Formatter.java:2520)
    at java.util.Formatter.format(Formatter.java:2455)
    at java.lang.String.format(String.java:2940)
    at StringFormatting.main(StringFormatting.java:45)
```

# What are exceptions?

- Exceptions are **objects**
- Exceptions are made and **thrown** when something bad happens
- We can **catch** exceptions in our programs so that we can make the bad things better
- Some exceptions **have to be caught**
  - Known as **checked** exceptions
- Some don't have to be caught (the ones you've seen)
  - Known as **unchecked** exceptions
- Dealing with files requires handling **checked** exceptions
- But we'll look at **unchecked** ones first

# Unchecked exceptions

- Unchecked exceptions are common e.g.:
  - You try and access an element of an array that doesn't exist
  - You try and convert "Hello" into an integer
  - ....
- Normally, they cause some red text to appear, and your program to stop
- You can also make them happen on purpose using the **throw** command

```
public class Throw {  
    public static void main(String[] args) {  
        System.out.println("About to throw an unchecked exception");  
        throw new ArrayIndexOutOfBoundsException();  
    }  
}
```

### **Output:**

```
About to throw an unchecked exception  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException  
    at Throw.main(Throw.java:5)
```

# Throw your own

## Output:

About to throw an unchecked exception  
Exception in thread "main" MyException  
at Throw.main(Throw.java:5)

- You can also make your own Exception (...exceptions are objects...)
- To make an unchecked exception, you **extend** the **RuntimeException** class

```
public class MyException extends RuntimeException {  
}
```

```
public class Throw {  
    public static void main(String[] args) {  
        System.out.println("About to throw!");  
        throw new MyException();  
    }  
}
```

# Catching

- Exceptions become really useful once you know how to **catch** them
  - Remember: all exceptions **can** be caught. Some (checked ones) **have to be caught**.
- Catching an exception detects that it has happened and allows you to act accordingly.
- Achieved using `try / catch / finally` blocks

```
try {  
    // code A  
} catch(<ExceptionType> <name>) {  
    // if code A causes an exception  
} finally {  
    // some code done regardless  
}
```



# Simple example

```
import java.util.Random;
public class ArrayException {
    public static void main(String[] args) {
        int[] x = new int[10];
        Random r = new Random();
        int pos = r.nextInt(20);
        System.out.println(x[pos]);
    }
}
```

- This code will (sometimes) cause `ArrayIndexOutOfBoundsException` to be thrown (when `pos >= 10` (the length of the array))
- We can **catch** this exception...

```
import java.util.Random;
public class ArrayExceptionCaught {
    public static void main(String[] args) {
        int[] x = new int[10];
        Random r = new Random();
        try {
            int pos = r.nextInt(20);
            System.out.println(x[pos]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Too big");
            e.printStackTrace();
        } finally {
            System.out.println("This happens whatever");
        }
    }
}
```

# Sample Exception III

- Now if  $\text{pos} \geq 10$  the Exception is caught (by the catch statement) and we do something (in this case, prints “Too big” and the stack trace)
  - If  $\text{pos} < 10$  we don't enter the catch block
  - Either way we enter the finally block
- 
- In this case, we could do the same with if / else
  - But in many cases try / catch is much neater (and compulsory!)

# Aside - Random

- In the last example we used Random
- Random is a Java class that provides the capability for generating (pseudo)random numbers.
- We created an instance of the class:
  - `Random r = new Random();`
- And then called the `nextInt(int max)` method to generate random integers between 0 and max-1
- It has methods for generating other types (e.g. doubles)

# COMPSCI4039: Programming

FileIO and Exceptions

# A checked exception - FileNotFoundException

- FileReader is a Java class that allows you to read characters from a file
- Here's an excerpt from the Java API
- It tells us that the constructor throws FileNotFoundException and we therefore have to **catch** it

## FileReader

```
public FileReader(String fileName)  
    throws FileNotFoundException
```

Creates a new FileReader, given the name of the file to read from.

### Parameters:

fileName - the name of the file to read from

### Throws:

**FileNotFoundException** - if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

# FileReader

FileReader object is created in the try.  
Catch catches the Exception (FileNotFoundException) that the FileReader constructor might throw

```
import java.io.FileNotFoundException;
import java.io.FileReader;
public class FR {
    public static void main(String[] args) {
        try {
            String fN = "C:/Users/Chris/Documents/students.csv";
            FileReader fr = new FileReader(fN);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

printStackTrace() is a useful method that Exceptions have for debugging

- It is the constructor that **throws** the exception
- So we have to put the code that makes the FileReader (new FileReader(filename) etc) into a **try** block

# Closing the `FileReader`

- `FileReader` objects need to be **closed** when we're finished with them
- It is good practice to do this in a **finally** block: it gets closed whatever happens.
- The API tells us that the `close` method of `FileReader` can throw an `IOException` which we will need to catch
- We do this in a separate `try` `catch` block within the `finally` of the first `try` `catch` block
- ...and we need to create the `FileReader` reference outside the first `try` `catch` block so that it is in scope



Make the  
reference here so  
it is visible in  
finally

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class FR2 {
    public static void main(String[] args) {
        FileReader fr = null;
        try {
            String fN = "C:/Users/Chris/Documants/students.txt";
            fr = new FileReader(fN);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                fr.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Try and close the  
FileReader, catching  
the IOException that  
might be thrown.

# FileReader

- We can now do the actual file reading
- But first, a couple more useful things about Exceptions...

# Catching 'higher' exceptions

- `FileNotFoundException` inherits from `IOException` which inherits from `Exception`, which also inherits from `Throwable`
- Anywhere where we need to catch `FileNotFoundException`, we could catch any of these types instead, and it would be caught (this is an example of **polymorphism**)

## **Class `FileNotFoundException`**

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.io.FileNotFoundException
```

example...

```
        fr = new FileReader(fN);  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            fr.close();  
        } catch (Throwable e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- This is from the end of our last program. We've swapped:
  - FileNotFoundException for its super class, IOException
  - IOException (second catch statement) for its super-super class Throwable
- Both of the exceptions we need to catch could be caught as IOException (FileNotFoundException is a subclass of IOException)

- This solution is not as neat - it's good practice to catch exceptions individually

```
import java.io.FileReader;
import java.io.IOException;
public class FR3 {
    public static void main(String[] args) {
        FileReader fr = null;
        try {
            String fN = "C:/Users/Chris/Documents/students.txt";
            fr = new FileReader(fN);
            // read file
            fr.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

This will catch Exceptions thrown by either the FileReader constructor or the close() method.

# Delegating catches

- If you don't want to deal with the exception, you can delegate it up to whatever called your method.
- This is achieved by adding **throws <Exception1>, <Exception 2>, ...** into the method declaration:

```
public void myMethod(String a) throws IOException, ArrayIndexOutOfBoundsException {
```

- This means you can get rid of all the exception handling in our FileReader by defining our main to throw the exceptions...

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class FR4 {
    public static void main(String[] args) throws FileNotFoundException, IOException{
        String fN = "C:/Users/chris/Documents/students.txt";
        FileReader fr = new FileReader(fN);
        // some reading
        fr.close();
    }
}
```

- Defining methods that throw exceptions can be useful
- In this case, it's just a bit lazy! (useful for quick prototyping)
- Q: the code above is a bit redundant - what could we remove?

# Delegating catches 2

```
public class Delegation {  
    public static void main(String[] args) {  
        try {  
            myMethod();  
        } catch (MyException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void myMethod() throws MyException {  
        // some code  
        if (new Random().nextInt(10) <= 7) {  
            throw new MyException();  
        }  
    }  
}
```

Because myMethod can throw a MyException, it has to be in a try catch block.

A MyException can be thrown in this method. It doesn't want to deal with it, so throws it up to the calling method (main).



# Order is important

- Because of polymorphism, multiple catch statements could potentially catch the same Exception:
- In such cases, which one will catch it?
- E.g. Where will the FnF exception be caught?

```
public class ClassQ {  
    public static void main(String[] args) {  
        try {  
            throw new FileNotFoundException();  
        } catch (IOException e) {  
            System.out.println("IO");  
        } catch (FileNotFoundException e) {  
            System.out.println("FnF");  
        }  
    }  
}
```

Exceptions are caught by the first (top to bottom) catch statement that they can be caught by

In this case, Java will not compile because all FnF exceptions will be caught by the IOException making the line that prints FnF **unreachable**

# COMPSCI4039: Programming

FileIO and Exceptions

# Reading from a file

- Now we've sorted out Exception handling (it's not just a file thing...) we can read from the file.
- FileReader allows us to read individual characters
- This is tedious - we use a Scanner to read in complete lines
- We pass the FileReader to the Scanner when we make it...

```
// make a filereader object
fr = new
FileReader("/Users/simon/Desktop/students.csv");
// make a scanner around the filereader
Scanner s = new Scanner(fr);
```

# Files and paths

- When we create a `FileReader` we will pass it the name of the file.
- This needs to include the complete **path** to the file
- In my case, the file is on my desktop: `/Users/simon/Desktop/fileName`
- On the lab machines (Windows) it might be something like this:
  - `H:\aDirectory\anotherDirectory`
- And remember that `\` is a special character so to get a `\` you need to use `2!`
  - `String fileName = "H:\\aDirectory\\anotherDirectory\\etc";`

# Reading from a file

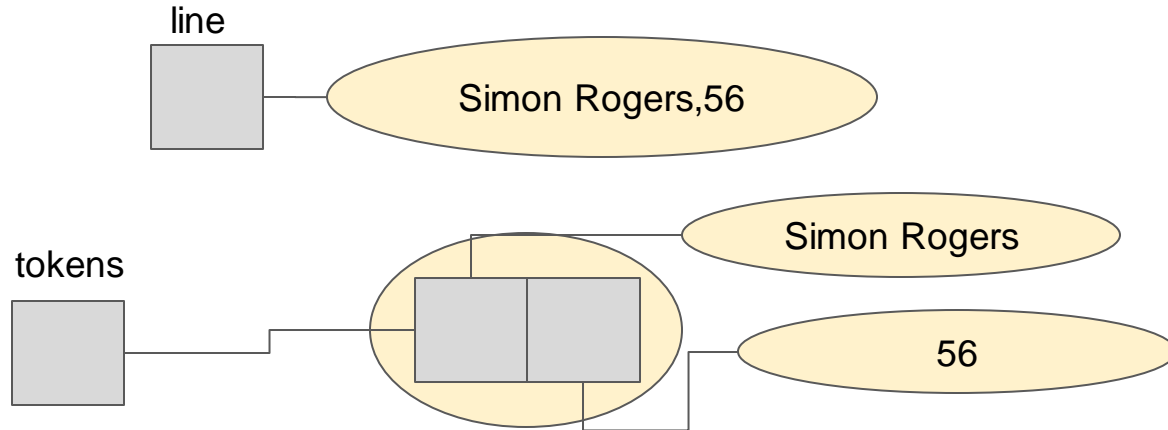
- We then loop while there are more lines to be read:

```
// Loop until no lines left
while(s.hasNextLine()) {
    // get the next line
    String line = s.nextLine();
```

# Reading from a file

- Each line has the following format:
  - <name>,<grade>
- We will use the `String.split` method which splits a `String` into an array of smaller strings
- If we ask it to split using a comma, it will give an array of all of the comma separated elements
- E.g.
  - `String line = "Simon Rogers,56";`
  - `String[] tokens = line.split(",");`

- o `String line = "Simon Rogers,56";`
- o `String[] tokens = line.split(",");`



# Reading from a file

- Finally, we need to turn the grade into an `int` (it's currently a `String`)
- The **static** `parseInt()` method from the `Integer` class will do this:
  - `int grade = Integer.parseInt(tokens[1]);`
- We can now make a new `Student` object from the name and grade:
  - `new Student(tokens[0],grade);`
- And store it into the array (also increasing the number of students)
  - `students[nStudents++] = new Student(tokens[0],grade);`



# Putting it all together

- See `LoadStudents.java`

## Overview:

- To read from the file we need a `FileReader` object
- We give this to a `Scanner` to get the contents line by line
- We then process each line (I used `String.split`, you could also use another `Scanner`)
- Around this process we need all of the exception handling things.
- We needn't have done things line by line - could use any `Scanner` methods...e.g. `next()`, `nextInt()`, etc

# COMPSCI4039: Programming

FileIO and Exceptions

# Writing to a file

- Say we now want to sort the students and write them (sorted) to a new file
- Our Student object has a compareTo method (Comparable interface) so we can use `Arrays.sort`
- To write to a file we use a `FileWriter` object...
- The same Exception handling is required
- Once we've done that, we can write any `String` to the file using the `FileWriter`'s `write()` method.
- See `LoadStudents2.java`
  - [ had to add a `getName()` method to `Student`]

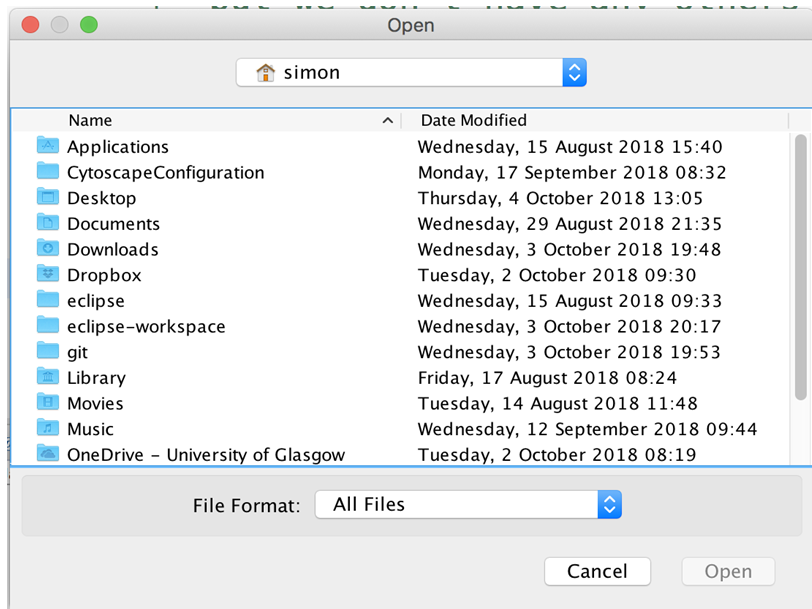
# BufferedWriter

- You might see examples that use a `BufferedWriter` to write to the file
- It can be beneficial to put things in a buffer (temporary storage) and only write to the file once there is enough there
- This is what `BufferedWriter` does.
- You can force it to write by **flushing** the buffer (`bw.flush()`)
- Data also written when:
  - Buffer is full
  - `BufferedWriter` is closed

```
fw = new FileWriter(outName);  
BufferedWriter bw = new BufferedWriter(fw);  
// maybe a loop or something  
bw.write(newLine);
```

# JFileChooser

- What if we want to allow the user to specify the file?
- We could ask them to type in the filename
- Or we could use a JFileChooser...



# JFileChooser

- JFileChooser is part of the Java Swing GUI library.
- It includes both open and save windows
- Very easy to use:
  - `JFileChooser fc = new JFileChooser();`
  - `int returnVal = fc.showOpenDialog(new JFrame());`
- This will pop up a JFileChooser dialog box
- `returnVal` tells you if a file was chosen or not (0 = yes)
- Access the File object using `fc.getSelectedFile();`
- **See ChoosingFiles.java**