GitLab so far

- **152** students registered
- **107** SSH keys uploaded
- **60** forks of lab 1
- **4** students pass all JUnit lab 1 tests
  - Harris Uddin
  - Euan Aitken
  - Jake Barr
  - Alexander Young
- Harris Uddin: labs **1, 2, 3** tests pass!

- **Windows**
  - ***Create SSH keys using Eclipse***
    - Learning Materials -> GitLab documentation -> Getting started – Windows and Mac OSX users
    - ***"Importing Maven projects has encountered a problem… Can't rename …"***
- **Mac OSX**
  - Follow the ***Windows instructions*** for generating SSH keys
    - Learning Materials -> GitLab documentation -> Getting started – Windows and Mac OSX user
    - If you've tried following the Linux instructions, delete theose SSH keys first.

Learning Materials -> GitLab documentation -> Troubleshooting GitLab/Eclipse

Thank to:
Euan Aitken, Alexander Markou, William Thorenfeldt, Jessica Blades

Software Development 3 (F27SG)

Lecture 2

IO, Strings & Exceptions

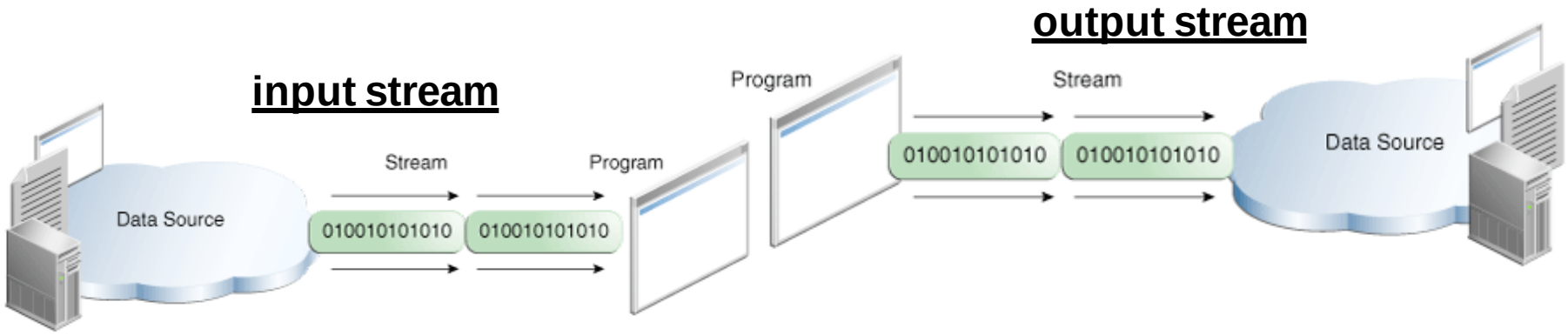
Rob Stewart

Outline

- By the end of this lecture, you should
  - be familiar with IO **streams**
    - byte and character streams
    - handle streams in Java
      - **From**: keyboard, text files,...
      - **To**: screens, speakers,...
  - understand **exceptions**
    - **Handling** exceptions in Java
    - **Testing for** exceptions with JUnit
  - be able to perform basic **string manipulations**

IO Streams

- An IO stream flows either from
  - an ***input*** source: disk, USB, ...
  - to an ***output*** destination/sink: screen, another program,...
- A stream is an *ordered sequence* of data
  - a program **reads** from an *input stream*
  - and **writes** to an *output stream*



Standard IO

- There are **three** streams that are often called standard IO streams.
- In Java, these are:
  - The **System.out** to write to *standard output*
    - often the screen
    - e.g. **System.out.println(“hello”);**
  - The **System.in** to read from *standard* input
    - often the keyboard
  - The **System.err** to write *standard error* messages
    - e.g. **System.err.println(“an error”);**

Streams

- The **java.io** package contains most of the Java classes for handling streams
- A program that wants to use these classes must
  import java.io.*;
- Or selectively import the required classes, e.g
  import java.io.Reader;
- Now we'll look at:
  - **byte streams** first
  - Then … **character streams**

Byte Streams

- A **byte stream** manages bytes of **raw binary** data
- Byte streams have any number of uses:
  - e.g. objects, music, graphics, file compression
- **System.out** and **System.err** are output byte streams
  - Both of them have **print** and **println** methods
- **System.in** is an **input byte stream**

Character Streams

- A **character stream** to handle 16-bit **Unicode** characters
  - Unicode contains more than 110,00 characters
  - Stream consists of a series of Unicode characters

Reading text

Reading text from:

- Keyboards: **Scanner** class
- Text files: **FileReader** class

Reading from Keyboard

- To read from the keyboard: use **java.util.Scanner** class
  - with **System.in** as parameter

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

- Some commands:
  - **sc.hasNext()** // checks if more to read
  - **String s = sc.next();** // next word
  - **String s = sc.nextLine();** // the next line
  - **int i = sc.nextInt();** // next integer

Reading a Text File (1)

- We will first discuss how to **read a text file**
- To read a file we must:
  - **open** the (stream from the) file
  - **read** the data (item by item) into variable(s)
  - **close** the (stream to the) file
- To read/write text we use ***character streams***

Exercise

- I would like to read from a *text* file, which stream should I use?
  - Byte stream
  - Character stream

www.socrative.com / `student login' / room: SD32019

Exercise

- I would like to read from a text file, which stream should I use?
  - Byte stream
  - Character stream

Reading a Text File (2)

- To achieve this we can use a **FileReader**
- Reading from a file may fail
  - ***Ideas why?***
- As a result of problems with file operations, **exceptions** may be raised

Exceptions

- An **exception** is an unexpected event occurring during execution of a program
- Many of the IO methods throw exceptions
  - e.g. the given file name does not exists.
  - **IOException** subclasses e.g.:
    - FileNotFoundException, FileSystemException
  - in most cases they throw an **IOException**
- **IOException** is most common for input streams
- You can explicitly throw an exception, e.g.

```
throw new IOException(“something failed");
```

Exceptions

- Any method using these exception classes must either
  1. be declared to **throws IOException**, or
  2. **catch** and handle the exception itself
- **An uncaught exception terminates the running program**

Handling Exceptions

- There are two ways of handling exceptions in a method
  - throw the exception

```
public void mymethod() throws IOException{ ...}
```

    - this must then be handled my a method calling mymethod()
  - catch the exception

```
try {
   <code>
} catch(IOException e) {
    <error code (can use e here)>
}
```

Exercise

- What happens to an uncaught exception?
  - It crashes the program
  - It prints an error and continues
  - Nothing

www.socrative.com / `student login' / room: SD32019

Solution

- What happens to an uncaught exception?
  - It crashes the program
  - It prints an error and continues
  - Nothing

Exercise

- Will this code compile:

```
public void test() {
   throw new Exception(“test");
}
```

www.socrative.com / `student login’ / room: SD32019

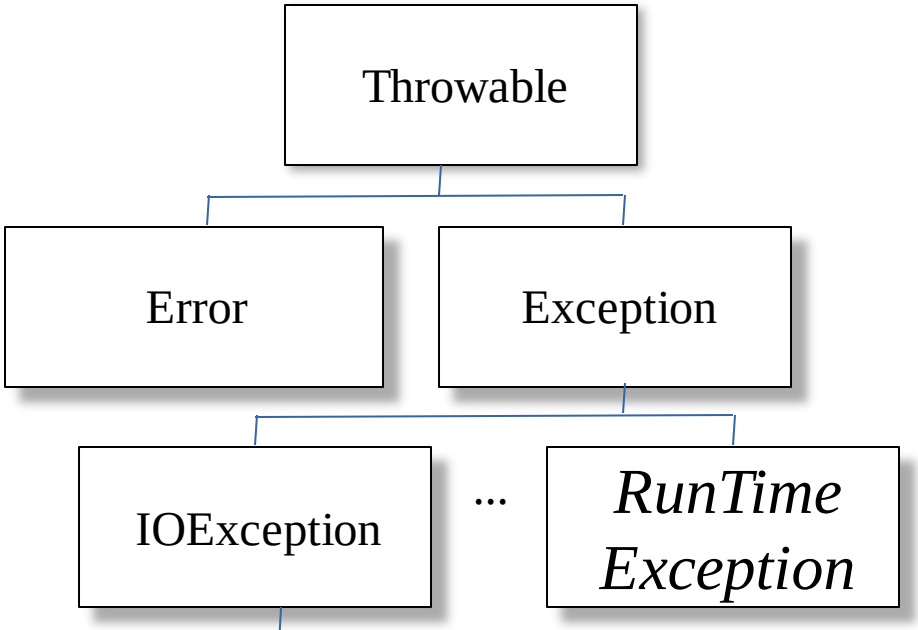
Exercise

- NO, you must declare that an exception is thrown in the signature

```
public void test() throws Exception {
  throw new Exception("test");
}
```

Top of the Exception Hierarchy

- Exceptions are organised in a hierarchy
- If you catch an exception you will also catch any of it's children
- You can also create your own exception at any point in the hierarchy, e.g




```
public class MyIOException extends IOException{
    public MyIOException(String message){
        super(message)
    }
}
```

- You typically want to invoke the parent constructor in your exception

Testing exceptions with Junit

- @Test indicates that a method is a test method
  - This will not work directly for exceptions

- We test that an exception **<name>** is thrown with the following annotation:
  - **@Test(expected = <name>.class)**

Junit exercise

- Which of these JUnit test will succeed?

```
@Test
public void testException1() throws Exception {
 throw new Exception("test");
}
```

```
@Test(expected = Exception.class)
public void testException2() throws Exception {
  throw new Exception("test");
}
```

```
@Test(expected = Exception.class)
public void testException3() throws Exception {

}
```

www.socrative.com / `student login' / room: SD32019

Junit exercise

- Which of these JUnit test will succeed?

```
@Test
public void testException1() throws Exception {
 throw new Exception("test");
}
```

(error)

```
@Test(expected = Exception.class)
public void testException2() throws Exception {
  throw new Exception("test");
}
```

```
@Test(expected = Exception.class)
public void testException3() throws Exception {

}
```

Reading a Text File (3)

- A **FileReader** accepts a path/name of the file to be read:

```
FileReader fr = new FileReader("path/filename.txt")
```

- A **FileReader** reads a stream of bytes an converts them into characters (text)
- A **BufferedReader** is suitable for reading lines of text
  - **i**t accepts a **FileReader**

```
BufferedReader br = new BufferedReader(fr)
```

Reading a Text File (4)

- A **BufferedReader** has several methods, e.g
  - **readLine()** returns a String of the next line of text
  - **ready()** returns **true** if there is anything left to read
  - **close()** closes the stream

- We will now show a simple method which prints every line of a file...

Reading a Text File (5)

```
public void readTextFile(String filename){
  try{
    FileReader fr = new FileReader(filename);
    BufferedReader br = new BufferedReader(fr);
    String line;
    while(br.ready()){
          line = br.readLine();
          System.out.println(line);
      } // end of while loop
     br.close();
  } // end of try clause
….
```

Reading a Text File (6)

- We also need to handle the exceptions which are thrown
- Handling exception is in the **catch** clause below
- By catching an **Exception** object **all** exceptions are handled
  - we can also handle ***different types of exceptions separately***
  - e.g. catch **IOException** (subclass of **Exception**)
  - each type of exception will be in a separate **catch** clause
- The **getMessage()** method of the **Exception** class returns a String with the details of the exception

```
catch(Exception e){ // will catch any exception
      System.out.println(e.getMessage());
 } // end of catch clause
} // end of method
```

The String Class

- The String class offers many methods which may be useful
  - **s.length()** returns the length of string s
  - **s.charAt(i)** returns the char at index i
  - **s.equals(t)** determines if strings s and t are equal
  - **s.concat(t)** returns a string of s followed by t
  - **s.split(t)** returns a string array of substrings of s separate by t.
  - **s.trim()** deletes leading and trailing whitespace
    - Note that for our word count we should really have written
      words = line.trim().split(" ");
- Java Strings are *immutable (final)*
  - operations such as **concat** and **trim** does not change the given string but returns a new copy with operations applied to it

Splitting Strings

- **readLine()** returns complete line of the text file
- **split** method on a string creates string array of substring
- We have to provide a **delimiter**
- To illustrate
  - String s = “a-b-c-d”
  - Split each letter (“-” is a delimiter)
  - **s.split(“-”)** will return the array **{“a”,”b”,”c”,”d”}**

Counting the number of Words (1)

```
public int count(String filename) throws Exception{
      FileReader fr = new FileReader(filename);
      BufferedReader br = new BufferedReader(fr);
      String line;
      String[] words;
      int count = 0;
      while(br.ready()){
          line = br.readLine();
          words = line.split(" ");
        count +=  words.length;
     } // end of outer while loop
     br.close();
     return count;
} // end of method
```

The StringBuffer Class

- A String cannot be changed
- The **StringBuffer** class is more efficient when the string is frequently changed

```
StringBuffer sb1 = new StringBuffer("a string");
StringBuffer sb2 = new StringBuffer();
```

- It has a number of operations, including
  - **sb1.append(sb2)** which appends sb2 to the end of sb1
  - **sb1.toString()** which returns the string representation
  - **subString**, **charAt**, …

Writing Text Files

- Writing string to text file similar to reading a file
- Create a **FileWriter** object
  - the FileWriter accepts a filename as argument
- Create a **BufferedWriter** object
  - With the FileWriter as argument
- **Write** the string to the BufferedWriter object
- **close** the stream

Writing Text Files

```
public void
writeTextFile(String filename,String content){
try {
    FileWriter wr = new FileWriter(filename);

    BufferedWriter bw = new BufferedWriter(wr);
    bw.write(content);
    bw.close();
  } catch(Exception e) {
     System.out.println(e.getMessage());
  }
}
```

Serialising Objects

- Data files are of **bytes** (binary) rather than **characters**
- We can save an **entire object** to a file
- When an object is saved, all referenced objects are also written to the file
- Must be ***serialisable***,
  - the class has to implement the **java.io.Serializable** interface

```
import java.io.Serializable;
public class MyClass implements Serializable { .... }
```

Writing Objects

- **ObjectOutputStream** to write serialisable object to file
- The constructor needs a **FileOutputStream**
  - takes a filename as argument
- **writeObject(obj)** to write the object
  - in addition, you need to handle the exceptions...

```
FileOutputStream fos = new FileOutputStream(filename);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(object);
oos.close();
```

Reading Objects

- **ObjectInputStream** to read a serialisable object to a file
  - this needs a **FileInputStream**, which needs a filename
- Write the object with **readObject()** method
  - returns an **Object**
  - you ***cast*** this object to the correct type (more next week)
- E.g. a serialisable object of class **MyClass** is read as:
  - (in addition, you need to handle the exceptions…)

```
FileInputStream fis = new FileInputStream(filename);
ObjectInputStream ois = new ObjectInputStream(fis);
MyClass m = (MyClass) ois.readObject(); // casted to MyClass
ois.close();
```

Resources

- Online resources
  - Exceptions
    docs.oracle.com/javase/tutorial/essential/exceptions/
  - IO
    docs.oracle.com/javase/tutorial/essential/io/
  - Strings
    docs.oracle.com/javase/tutorial/java/data/strings.html

Wrapper Classes

- Sometimes we want to convert a string to a primitive type
  - e.g. if we want to read in a price of something, which is currently a **String**, we may want to use an **int** to perform arithmetic operations on it
- A **wrapper class** wraps such a primitive type.

Wrapper class	Primitive type
Character	char
Integer	int
Double	double

- Wrapper classes provide utility methods over primitive values, e.g:

```
int val = Integer.parseInt(string);

double d = Double.parseDouble(another_string);
```

Autoboxing

- Java will automatically convert a primitive to it's wrapper class when required, and this is called **autoboxing**.
- E.g.:

```
Integer i = 1;
Character c = 'c';
```

Unboxing

- **Unboxing:** converting wrapper class to it's primitive
- Automatically applied when the wrapper class is:
  - passed as an argument expecting the primitive
  - assigned to a value expecting the primitive

```
public Integer myfun(int arg){ .... }
Integer i = 10; // autoboxing
int x = myfun(i) // unboxing
```

autoboxing or unboxing

1. Integer i = 5;

   Autoboxing

   converts primitive to wrapper class

2. Integer i = …;

   int j = i +5;

   Unboxing

   converts wrapper class (i) to its primitive

Exercise: autoboxing or unboxing

- Which is **unboxing** and which is **autoboxing**?
  - Integer i = 5;
  - Double d = 4.5;
  - double d = 3.2;
  - int i = 2;

www.socrative.com / `student login' / room: SD32019

Solutions:

- Autoboxing examples:
  - `Integer i = 5;`
  - `Double d = 4.5;`
  - `double d = 3.2;`
  - `int i = 2;`

www.socrative.com / `student login' / room: SD32019

IO Space Race

Summary

- Introduced basic **IO streams** concept and how streams are handled in Java
  - keyboard, text files and objects
- Introduced **exceptions** and exception **handling**
- **String** and **StringBuffer** classes
- **Wrapper** classes, **autoboxing** and **unboxing**

Next lecture: complexity