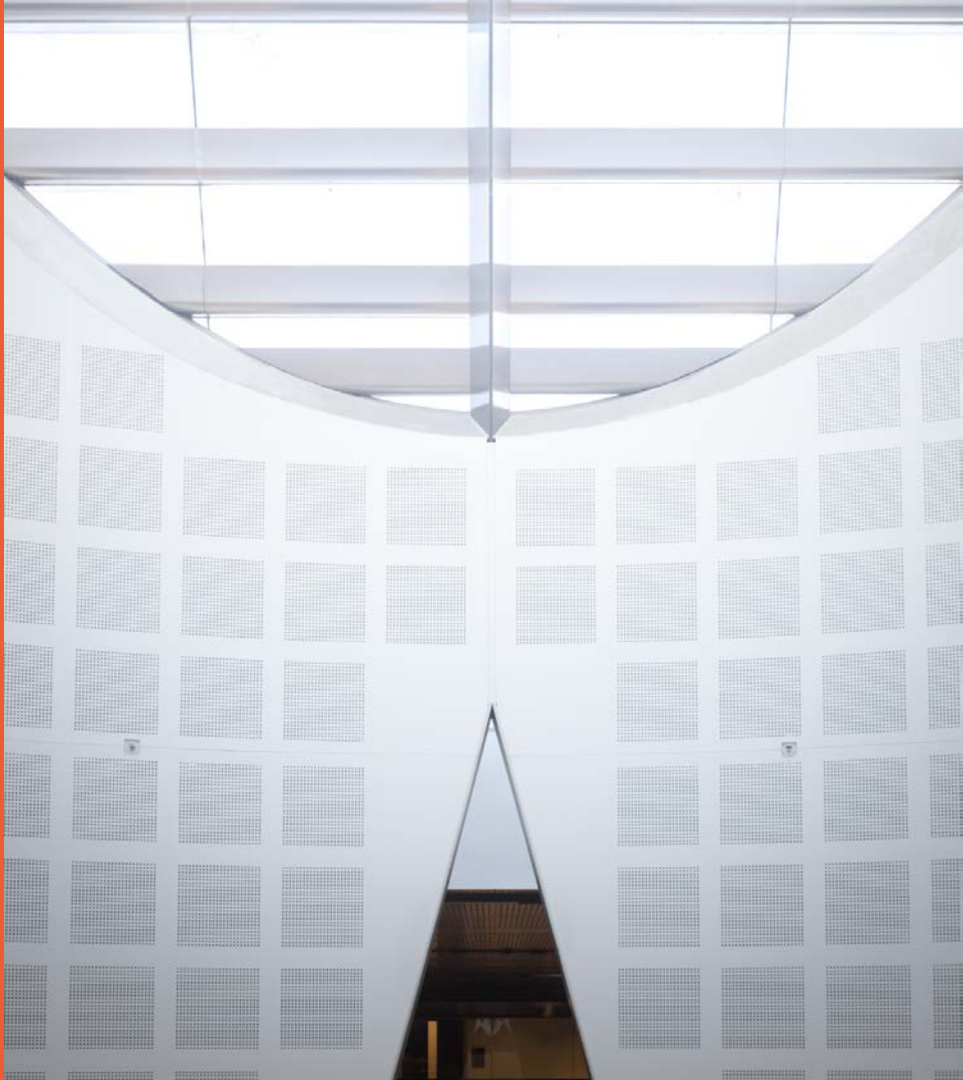
COMP9103: Software Development in Java

W8: Exception & File IO

Presented by
Dr Ali Anaissi
School of Computer Science

THE UNIVERSITY OF SYDNEY



Exception

THE UNIVERSITY OF SYDNEY

Exception

- An exception represents an abnormal condition in the program logic
- For example,
  - Attempting to divide by zero
  - Attempting to access an invalid array position
  - Trying to invoke a method using a null reference
  - Many others
- An exception that is not explicitly dealt with causes the program to crash

Exception

– The usual behavior on runtime errors is to abort the execution:

```
class TestExceptions1 {
        public static void main(String[] args) {
              String s = "Hello";
              System.out.print(s.charAt(10));
        }
}
```

```
$ java TestExceptions1
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: String index out of range: 10
at java.lang.String.charAt(String.java:499)
at TestExceptions1.main(TestExceptions1.java:11)
```

Exception

- The exception can be trapped:

```
class TestExceptions2 {
        public static void main(String[] args) {
                String s = "Hello";
                try {
                        System.out.print(s.charAt(10));
                } catch (Exception e) {
                System.out.println("No such position");
                }
        }
}
```

```
$ java TestExceptions2
No such position
```

Trying Code and Catching Exceptions

- **`try` block**
  - A segment of code in which something might go wrong
  - Attempts to execute
    - • Acknowledges an exception might occur
- A `try` block includes:
  - The keyword `try`
  - Opening and closing curly braces
  - Executable statements, which might cause an exception

Trying Code and Catching Exceptions

- **`catch` block**
  - A segment of code
  - Immediately follows a `try` block
  - Handles an exception thrown by the `try` block preceding it
  - Can "catch" an `Object` of type `Exception` or an `Exception` child class
- **`throw` statement**
  - Sends an `Exception` object out of a block or method so it can be handled elsewhere

Trying Code and Catching Exceptions

- A `catch` block includes:
  - The keyword `catch`
  - Opening and closing parentheses
    - An `Exception` type
    - A name for an instance of the `Exception` type
  - Opening and closing curly braces
    - Statements to handle the error condition

Trying Code and Catching Exceptions

- If no exception occurs within the `try` block, the `catch` block does not execute

- Within a `catch` block, you might want to add code to correct the error

Exceptions

– It is possible to specify interest on a particularexception:

```
class TestExceptions3 {
    public static void main(String[] args) {
        String s = "Hello";
        try {
            System.out.print(s.charAt(10));
         } catch (StringIndexOutOfBoundsException e) {
           System.out.println("No such position");
        }
    }
}
```

```
$ java TestExceptions3
No such position
```

Exceptions

– We can add multiple catch blocks and a finally clause:

```
class MultipleCatch {
    public void printinfo(String sentence) {
          try {
                    // get first and last char before the dot
                   char first  sentence.charAt(0);
                   char last  sentence.charAt(sentence.indexOf(".") - 1);
                   String out  String.format("First: %c Last: %c",first, last);
                   System.out.println(out);
            } catch (StringindexOutOfBoundsException e1) {
                    System.out.println("Wrong sentence, no dot?");
            } catch (NullPointerException e2) {
                    System.out.println("Non valid string");
            }
          }
}
```

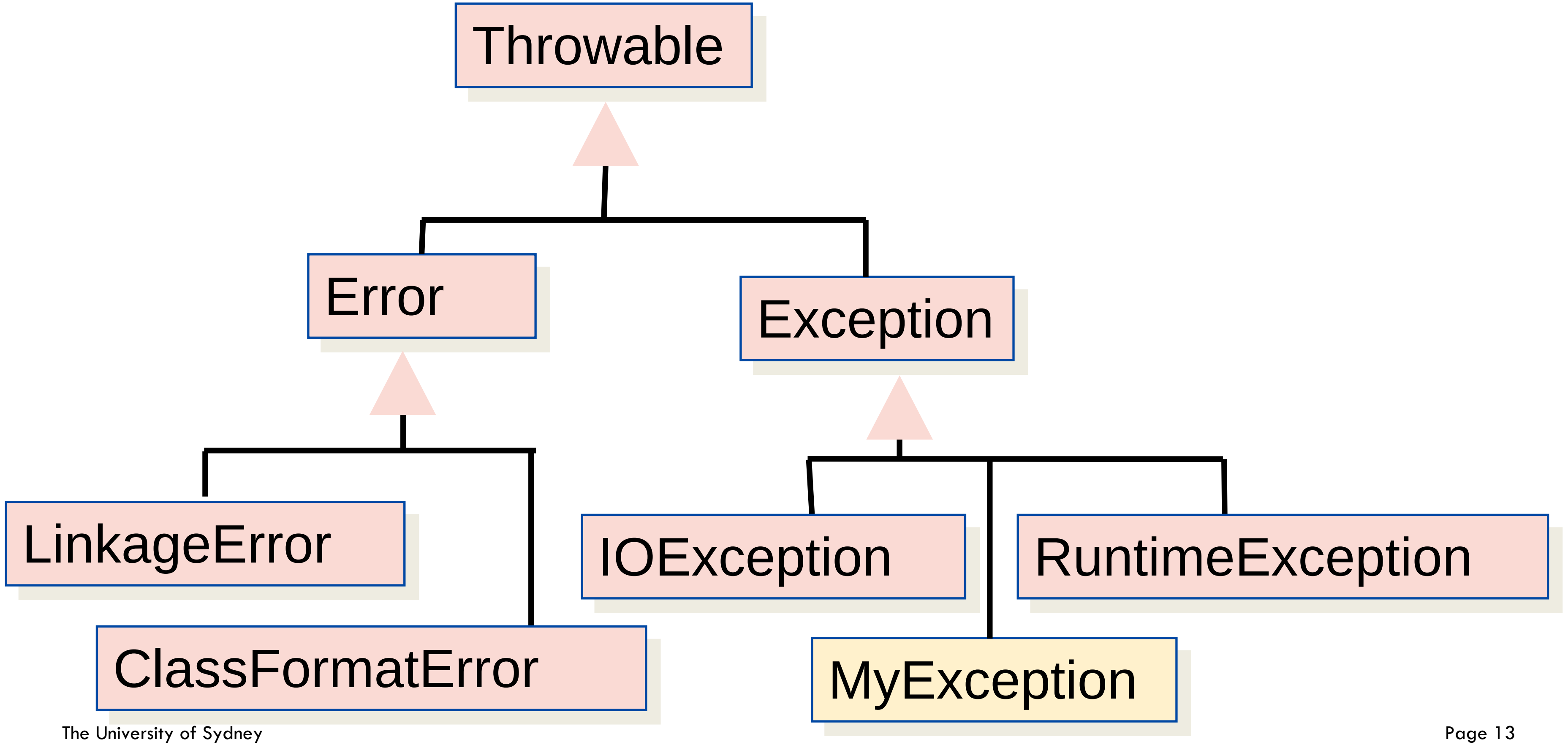
Checked vs. Unchecked exception

- Checked Exceptions: Compiler insists that you consider the possibility of these arising by having a catch block

- Unchecked Exception: Compiler doesn't insist on a catch block

- If a checked exception is not being caught in the method, that method's header must include a **throws** clause.

Exceptions – a Hierarchy of classes



BankAccount Example

```java
public class BankAccount {
    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }

    public boolean withdraw(double amount)
    {
        boolean returnVal = false;
        if (amount < balance) {
            balance -= amount;
            returnVal = true;
        }
        return returnVal;
    }
}
```

A Custom Exception Class

```
public class BankAccountException  extends Exception {
   public BankAccountException(String reason)
   {
      // pass to the Exception class constructor.
      super(reason);
   }
}
```

Generating and Throwing an Exception

```
public void withdraw(double amount) throws BankAccountException
{
    if (amount <= balance) {
        balance -= amount;
    } else {
        // generate an exception
        BankAccountException problem = new BankAccountException(
          "Not enough money to fulfil withdrawal request" );

        // throw the exception:
        throw problem;
    }
}
```

File I/O

THE UNIVERSITY OF SYDNEY

Introduction : I/O mechanisms

- All programs need I/O to communicate with the outside world
- I/O can be textual, graphical, through sensors, external devices, etc
- For desk computers: usually screen, keyboard, network, file system
- For embedded systems: all sorts of sensors and machinery

The Java IO package

- Provided as part of the JDK
  - Contains classes and interfaces to use when dealing with different types of input and output
  - Includes:
    - 12 Interfaces
    - 51 Classes
    - 16 Runtime Exceptions & 1 Error
- Required
  - **import java.io.*;**
  - **try/catch()** blocks to handle input / output exceptions

The File class

- Represents information *about* an existing particular file, or directory
  - Derived directly from class Object
- Restrictions
  - **Cannot write data to or read data from any files**
- Uses for this class:
  - List existing files and directories
  - Check for existence of files *and* directories
  - Used by other input/output objects that handle transferring of data
  - Many other housekeeping capabilities

Example: Creating a new instance of File

new File (String pathname)

Creates a new File instance by converting the given pathname string into an abstract pathname.

Example:

```
String fileName1 = "C:\\Program Files\\myFile.txt"
File f = new File(fileName1);
```

Example: File class methods

```
public static void main(String [] args)
{
    File f = new File(“myFile.txt");
    if( f.isFile() && f.canRead() && f.canWrite() )
       System.out.print("File exists and can be used");
    else
       System.out.print("File no good for purposes");
}
```

Example: Listing a Directory

```
public void listADirectory()
   {
      File dirList = new File("C:\\Users");
      String[] dList = dirList.list();
      for(int i = 0; i < dList.length; i++)
            System.out.println(dList[i]);
   }
```

String[] list(): Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

Text Files

THE UNIVERSITY OF SYDNEY

Text Data vs Binary Data

- All files are stored as binary data in the form of bytes
- **Text files** are human and text editor readable because they consist only of ASCII characters
- **Binary files** are not humanly readable because they use all 256 possible values of a byte.
- Binary files are readable only by programs that know how to interpret all the bytes

Text Files – Reading

- We can use a **Scanner** object to read the contents of humanly readable text files
- Create a new Scanner object and set up its input resource as the file object

```
File file-obj = new File(file-name);
Scanner inputFile = new Scanner(file-obj);
```

- Example:

```
Scanner inputFile = new Scanner ( new File ( "myFile.txt" ) );
String firstLine = inputFile.nextLine();
System.out.println("Data from file: " + firstLine);
```

Text Files – Writing

- We can use a **PrintWriter** object to write contents to humanly readable text files
- Class **PrintWriter** has methods **println(), print() and printf()** that are used in the same way as the methods in **System.out**
- Construct an output stream for writing data to a file

```
File fw = new File(file_name);
E.g. File fw = new File("myFile.txt");
```

- Create a new PrintWriter from an existing File object

```
PrintWriter out = new PrintWriter(fw);
PrintWriter out = new PrintWriter(new FileWriter(fw, true));
```

append the new contents to the end of the file

Text-File Output with *PrintWriter*

- If the named file (eg, myFile.txt) exists already, its old contents are lost.
- If the named file does not exist, a new empty file is created (and eg, named myFile.txt).
- Example

```
PrintWriter outputFile = new PrintWriter(new File("myFile.txt"));
outputFile.println("First line of output");
outputFile.print("Second line ");
outputFile.println("of output");
```

Opening and Closing files

– Scanner and PrintWriter constructors will ‘open’ files.

– All files must be closed when no longer needed to release resources.

```
inputFile.close();
outputFile.close();
```

Exceptions arising from File I/O

- In compiling and/or running programs which deal with files and streams, you may come across methods throwing exceptions:
  - IOException
  - FileNotFoundException
  - EOFException
  - ….
- These must be handled by your code

Example: File Input via Scanner class

File: NumberList.txt available for use

```
1.0
13.0 12.3 14.2
10.987 23.3
```

Calculate the average of double values in an input file

```
import java.util.*;
import java.io.*;
public class AverageInFile1 {
  public static void main(String[] args){
    try{
        File file = new File(args[0]);
        Scanner reader = new Scanner(file);
        double sum = 0.0;  // cumulative total
        int num = 0;         // number of values

        // compute average of values in input file
        while (reader.hasNextDouble()){
            sum += reader.nextDouble();
            num++;
        }
       if(num>0)
           System.out.println("Average of values in ” +
           args[0] + “is ” + sum / num);
    }

    catch (Exception e) {
          System.out.println("Error: "+e.getMessage());
    }
  }
}
```

import some other java classes

Get a filename from the command line, set it up for reading

Get numbers from the file, compute their average

What to do if something goes wrong, called as **exceptions** in Java

Example: Text-File Output with *PrintWriter*

```
import java.io.*;
import java.util.*;
public class TextFileO {
	public static void main(String[] args) {
	  try {
	    File fw = new File(args[0] + ".txt");
	    PrintWriter out = new PrintWriter(fw);
	    Scanner in = new Scanner(System.in);
	    while (in.hasNextInt()) {
	              out.printf("%d", in.nextInt());
	                   out.println();
	    }
	    out.close();
	    System.out.println("inputs written into file successfully!");
	    } catch (FileNotFoundException e) {
	           System.out.println("The file not found.");
	    }
	}
}
```

File and **PrintWriter** constructors may throw a ***FileNotFoundException***, which means that the file could not be created.

Terminal interaction

```
> java TextFileO fileout
1 2 3 4 5 6 q
```

output: fileout.txt

```
1
2
3
4
5
6
```

You can read this file with a file editor

Binary Files

Binary Files

- The way data is stored in memory is sometimes called the *raw binary format.*
- Data can be stored in a file in its raw binary format.
- A file that contains binary data is often called a *binary file*.
- Storing data in its binary format is more efficient than storing it as text.
- There are some types of data that should only be stored in its raw binary format. E.g. secure data transfer over the network

Binary Files

- Binary files cannot be opened in a text editor such as Notepad.
- To write data to a binary file you must create objects from the following classes:
  - `FileOutputStream` - allows you to open a file for writing binary data. It provides only basic functionality for writing bytes to the file.
  - `ObjectOutputStream` - allows you to write data of any primitive type or String objects to a binary file. Cannot directly access a file. It is used in conjunction with a `FileOutputStream` object that has a connection to a file.

Write to a Binary File

– An ObjectOutputStream object is wrapped around a FileOutputStream object to write data to a binary file.

```
FileOutputStream fstream = new    FileOutputStream("MyInfo.dat");
ObjectOutputStream outputFile = new  ObjectOutputStream(fstream);
```

– If the file that you are opening with the FileOutputStream object already exists, it will be erased and an empty file by the same name will be created.

– Once the ObjectOutputStream object has been created, you can use it to write binary data to the file.

Example:

```
import java.io.*;
public class WriteBinaryFile
{
          public static void main(String[] args)    throws IOException
         {
             Customer c = new Customer(“Tom”, 100,500);
             FileOutputStream fstream =   new FileOutputStream(“customer.dat");
             ObjectOutputStream out =    new ObjectOutputStream(fstream);
            out.writeObject(c)
            out.close();
       }
}
```

Appending Data to Binary Files

- The `FileOutputStream` constructor takes an optional second argument which must be a `boolean` value.
- If the argument is `true`, the file will not be erased if it exists; new data will be written to the end of the file.
- If the argument is `false`, the file will be erased if it already exists.

```
FileOutputStream fstream = new    FileOutputStream("MyInfo.dat",true);
ObjectOutputStream outputFile = new  ObjectOutputStream(fstream);
```

Read from a Binary File

- To open a binary file for input, you wrap a ObjectInputStream object around a FileInputStream object.

```
FileInputStream fstream = new FileInputStream("MyInfo.dat");
ObjectInputStream inputFile = new ObjectInputStream(fstream);
```

- Once the ObjectInputStream object has been created, you can use it to read binary data from the file.

Example

```
import java.io.*;
public class ReadBinaryFile
{
          public static void main(String[] args)    throws IOException
         {
              Customer c = null;
              FileInputStream fstream =   new FileInputStream(“customer.dat");
              ObjectInputStream in =    new ObjectInputStream(fstream);
              c = (Customer) in.readObject();
             in.close();
        }
       System.out.println( c ) ;
}
```

Serialization

– Serialization is the process of converting an object to a sequence of bytes, to allow future reconstruction of the object.

– Any attributes marked ‘transient’ will *not* be serialized

– After a serialized object has been written into a file, it can be read from the file and deserialized.

– Classes **ObjectInputStream** and **ObjectOutputStream** contain the methods for serializing and deserializing an object.

Serialization

– In order to make the previous example code works, Customer class must "implements Serializable"

```
public class Customer implements Serializable
 {
      private double creditCardBalance;
      private double chequeAccountBalance;
      private String name;
…
}
```

Questions?

THE UNIVERSITY OF SYDNEY