Helwan University

HCIlab
Human Computer Interaction Laboratory

Interaction Group

Faculty of Computer Science

Programming Language 2
Object oriented design using JAVA

Dr. Ayman Ezzat
Email: ayman@fcih.net
Web: www.fcih.net/ayman

Introduction

Unit 4: Input, output and exceptions

Outline

1. Introduction
2. Input and output streams
3. Exceptions
4. Declaring and handling exceptions
5. Other error-handling techniques

1. Introduction

In this unit, we aim to study:

- **Inputs and Outputs**
  - There are 2 ways for input/output:

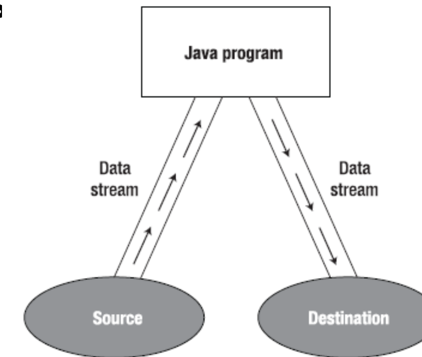
    (1) communicating with a sequential source or destination (such as a file, a keyboard or another computer).

    (2) using a graphical user interface (studied later).

- **Error Handling**
  - We show how to deal with error conditions in a Java program, by means of exceptions or otherwise;

2. Input and output streams

- **A stream** is essentially a **sequence of bytes**, representing a **flow of data** from a s
  - Sources and destinations include:
    - keyboard,
    - screen
    - various sorts of data files,
    - and between networked computers.



- **Any read or write is performed in three simple steps:**

  Step 1. Open the stream
    - **you need to define some objects here**

  Step 2. Until there is more data, keep reading in a read, or writing in a write.
    - **You need to use the methods of the objects defined in step1 to read the stream.**

  Step 3. Close the stream.

- **Note that**: using the statement

```
System.out.println("The program has started");
```

i.e. displaying information on the screen is immediately available for use without having to follow the above three steps.

2. Input and output streams

Reading text from the keyboard using Scanner class

Import
- java.util.* library that includes the Scanner class

Step1: define a Scanner object, ***sc*** or any name. To take input from the keyboard, use (System.in)

Step2: use the Scanner object to get data

Step3: close the Scanner when finished using it!

```
import java.util.*;
...
Scanner sc = new Scanner(System.in);
String name = sc.next();
int age = sc.nextInt();
System.out.println("you wrote:" + name+ "\n" +age);
sc.close();
```

2. Input and output streams

Reading text from a file using Scanner class

Import :
- java.util.* to use Scanner class
- java.io.* to use File class

Step1a: create a File object that points to your text file.

Step1b: define a Scanner object, ***sc*** or any name. To take input from the text file, use the file object you created in Step2a

Step2: use the Scanner object to get data

Step3: close the Scanner when finished using it!

```
import java.util.*;
import java.io.*;
...
File myFile = new File("C:/testing.txt");
Scanner sc = new Scanner(myFile);
String s = sc.nextLine();
System.out.println(s);
sc.close();
```

→ Note that for this code to work properly, we need to write some **error handling code** to handle the situation when the file is not found. We will deal with this later in this unit.

2. Input and output streams

Common Scanner methods

input methods:

s = *sc*.**next**()	Returns next “token” (i.e. "word“).
s = *sc*.**nextLine**()	Returns an entire input line as a String.
i = *sc*.**nextInt**()	Returns next integer value.
d = *sc*.**nextDouble**()	Returns next double value.
x = *sc*.**nextXYZ()**	Returns value of type XYZ (primitive value if possible), where XYZ is one of BigDecimal,BigInteger,Boolean,Byte,Float, or Short.

test methods (used for error checking and loops):

b = sc.**hasNext**()	True if another token is available to be read.
b = sc.**hasNextLine**()	True if another line is available to be read.
b = sc.**hasNextInt**()	True if another int is available to be read.
b = sc.**hasNextDouble**()	True if another double is available to be read.
b = sc.**hasNextXYZ**()	XYZ stands for one of the input types available above.

Source: http://leepoint.net/notes-java/summaries/summary-scanner.html

2. Input and output streams

Getting input using BufferedReader and another input stream reader

- The `BufferedReader` wraps another Reader and improves performance.

  Readers: (1) `InputStreamReader` to get input from keyb. as bytes and translates it to characters
  (2) `FileReader` to get input from files.

- The `BufferedReader` provides a `readLine` method to read a line of text.
  - Note that **`InputStreamReader`** doesn't provide a **`readLine`** method.
- Java also has a class called **BufferedWriter** that has a similar function for writing data.

Example1: reading input from keyboard using **InputStreamReader**

Step1: open an input stream

Step2: get data

Step3: close the input stream

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
String s = stdin.readLine();
System.out.println(s);
stdin.close();
```

Example2: reading input from a File using **FileReader**

Step1: open an input stream

Step2: get data

Step3: close the input stream

```
BufferedReader in = new BufferedReader(new FileReader("C:/test.txt"));
String s;
while ((s = in.readLine()) != null)
    System.out.println(s + "\n");
in.close();
```

2. Input and output streams

Writing text to a file using PrintWriter class

Import
- java.io.* to use the PrintWriter class

Step1a: create a File object that points to your text file.

Step1b: create PrintWriter object that points to your text file.

Step2: use the PrintWriter object to write text

Step3: close the PrintWriter when finished using it!

```
import java.io.*;
...
File myFile = new File("C:/aaa.txt");
PrintWriter pr = new PrintWriter(myFile);
pr.println("aaaaaa");
pr.println("bbbbbb");
pr.close();
```

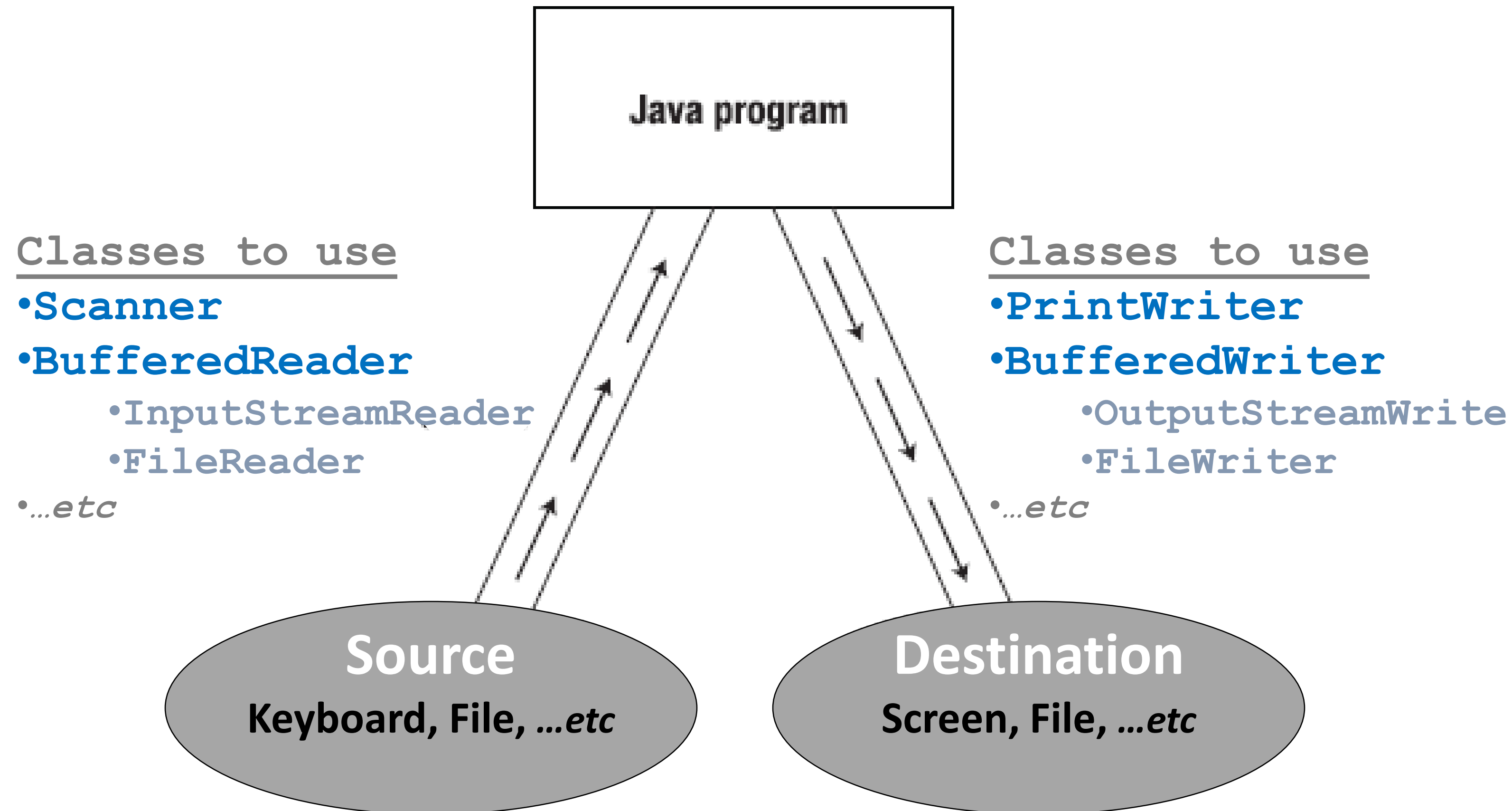
Notes:

- Any data in the aaa.txt file will be erased before writing the new data using `println` method.
- For this code to work, we need to write some **error handling code** to handle the situation when the file is not found. We will deal with this later in this unit.

If your program exits without closing the `PrintWriter`, the disk file may not contain all of the output.

2. Input and output streams

SUMMARY (1/2)



2. Input and output streams

SUMMARY (2/2)

READING	From Keyboard	From File

`Scanner`	`Scanner in = new Scanner(X);`
	`myString = in.next(); myInt = in.nextInt(); //etc`
	`in.close();`	

Replace X with:	`System.in`	`new File("C:/filename")`

`BufferedReader`	`BufferedReader in = new BufferedReader(X);`
	`myString = in.readLine();`
	`in.close();`	

Replace X with:	`new InputStreamReader(System.in)`	`new FileReader("C:/filename")`

WRITING	To Screen	To File

`PrintWriter`	`PrintWriter out = new PrintWriter(X);`
	`out.println("some text here");`
	`out.close();`	

Replace X with:	`System.out`	`new File("C:/filename")`

`BufferedWriter`	`BufferedWriter out = new BufferedWriter(X);`
	`out.write("some text here");`
	`out.close();`	

Replace X with:	`new outputStreamWriter(System.out)`	`new FileWriter("C:/filename")`
	OR instead of both classes, simply write: `System.out.println("some text here");`	

Appending to file

```
BW=new BufferedWriter(new FileWriter(FileName,true));
```

Selecting File Character Set

```
BW=new BufferedWriter(new OutputStreamWriter(new FileOutputStream(FileName,true),"UTF-8"))
```

Field and Description
ISO_8859_1 ISO Latin Alphabet No.
US_ASCII Seven-bit ASCII, a.k.a.
UTF_16 Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark
UTF_16BE Sixteen-bit UCS Transformation Format, big-endian byte order
UTF_16LE Sixteen-bit UCS Transformation Format, little-endian byte order
UTF_8 Eight-bit UCS Transformation Format

Storing Objects in Text files

```
Student Ahmed=new Student();
Ahmed.FullName="Ahmed Mohamed Ahmed";
Ahmed.age=12;
Ahmed.id=1;
Student Mohamed=new Student();
Mohamed.id=2;
Mohamed.age=19;
Mohamed.FullName="Mohamed Ibrahim";

PrintWriter NewFile=new PrintWriter("d:\\JavaFiles\\StoreStudent.txt");
NewFile.println(Ahmed.id + "@" + Ahmed.age+ "@" + Ahmed.FullName+ "@" );
NewFile.println(Mohamed.id + "@" + Mohamed.age+ "@" + Mohamed.FullName+ "@")
NewFile.close();
```

Special symbol to separate fields
Otherwise fields will be mixed together

How about storing emails in this case!!!
Another Symbol ~ for example ?

Sequence of writing object is very important

Reading Objects from text file

```
Scanner scan1=new Scanner(NewFile1);
Student []AllStudents=new Student[5];
int j=0;
while (scan1.hasNext())
    {
        AllStudents[j]=new Student();
        String Line=scan1.nextLine();
        String []Seprated=Tokens(Line, '@');
        AllStudents[j].id=Integer.parseInt(Seprated[0]);
        AllStudents[j].age=Integer.parseInt(Seprated[1]);
        AllStudents[j].FullName=Seprated[2];
        System.out.println("All student ID= " + AllStudents[j].id +
      j++;
    }
```

Loop till end of File

Separating line to tokens

Casting

Sequence of Reading

String Tokenizer

```
StringTokenizer st2 = new StringTokenizer("12@1234@Ayman Mohamed Ezzat@GPA=2.3", "@");

while (st2.hasMoreElements()) {
        System.out.println(st2.nextElement());
}
```

Output as strings

```
12
1234
Ayman Mohamed Ezzat
GPA=2.3
```

```
public static String[] Tokens(String Line,char seprator)
{
    String []Result=new String[10];
    String Word="";
    int ctr=0;
    for (int i=0;i<Line.length();i++)
    {
        if (Line.charAt(i)!=seprator)
        {
            Word+=Line.charAt(i);
        }
        else
        {
        Result[ctr]=Word;
        ctr++;
        Word=new String();
        }
    }
    return Result;
}
```

Concatenate charachters

Encryption/Decryption File

```
File MyFile=new File ("d:\\JavaFiles\\TestJava.txt" );
PrintWriter PW=new PrintWriter("d:\\JavaFiles\\Encrypt.txt");
Scanner scan=new Scanner(MyFile);
int i=1;
while (scan.hasNext())
{
    String Line=scan.nextLine();
    String EncLine=Encrypt(Line,1);
    PW.println(EncLine);
    System.out.println("Line "+ i + " - "+ Line);
    i++;
}
PW.close();
```

```
public static String Encrypt (String Word, int key)
{
    String result="";
    for (int i=0;i<Word.length();i++)
    {
        result+=(char) (Word.charAt(i)+key);
    }
    return result;
}
```

Reading address from the internet

```java
URL l=new URL("http://www.google.com");
Scanner xn=new Scanner(l.openStream());
while (xn.hasNext())
{
    System.out.println(xn.nextLine());
}
xn.close();
```

Open Stream

Process

Close

3. Exceptions

- **An exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions

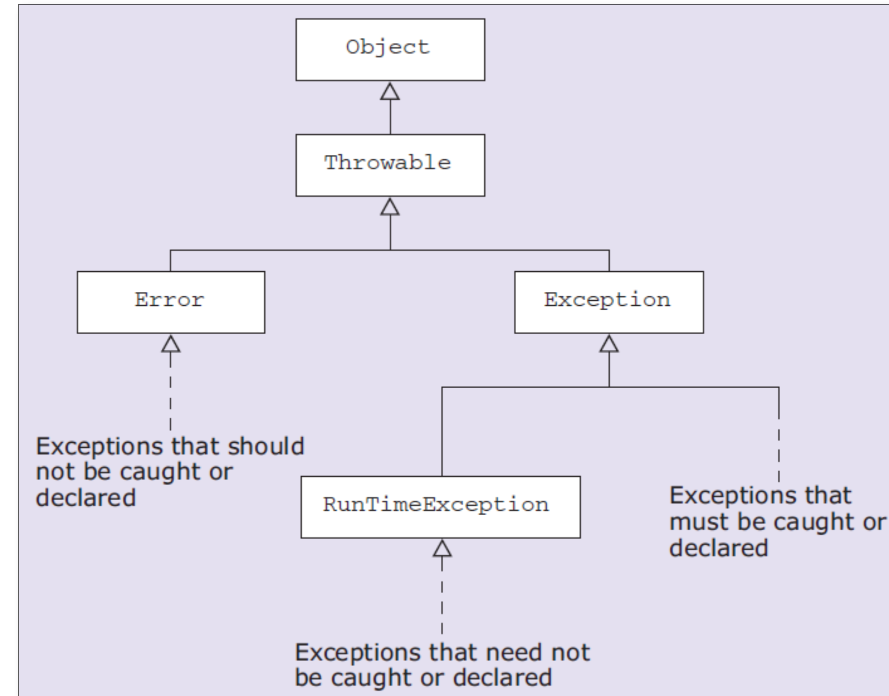
- Java's exception handling can be used to deal with potentially serious or unpredictable error conditions.
  - Examples of serious errors that could occur within a Java program are as follows:
    - a program tries to read past the end of a data file;
    - a program cannot find a specified input file;
    - a program tries to connect to a website using an invalid address;
    - a program runs out of memory;
    - a method tries to access an array element whose index is larger than the upper limit of the array;
    - an overflow condition occurs when the result of some arithmetic operation exceeds the limit for the primitive data types involved;
    - a program expecting to read a file of integers finds a string value in the file;
    - a method expecting an object reference encounters a null reference instead.

3. Exceptions

- Exception handling in its simplest form, it works like this:
  1. When Java system detects an **error**, it **stops the normal flow** of program execution.
     - *We say here the code **throws** an **exception**.*
  2. A special kind of **Java object**, known as an **exception object**, is created.
     - This object holds some information about what has gone wrong.
     - There are different kinds of exceptions for different sorts of error conditions.
  3. **Control is transferred** from the part of your program where the error occurred **to** an **exception handler**, which deals with the situation.
     - *We say here that the exception handler **catches** the **exception**.*
  4. The **exception object is available to the exception handler** code and can be used to help decide on the appropriate action.

3. Exceptions

- The **`Throwable`** represents the most general class for describing exceptions. The subclasses of **`Throwable`** can be divided into three main groups:
  - **(1) the `Error` class and its subclasses.**
    - The class Error describes exceptions that occur when some internal Java error has happened – **for example**, the Java system has run out of memory. Such errors are rare and there is little that a programmer can do about them.
  - **(2) The `Exception` class and its subclasses, *excluding* the `RunTimeException`.**

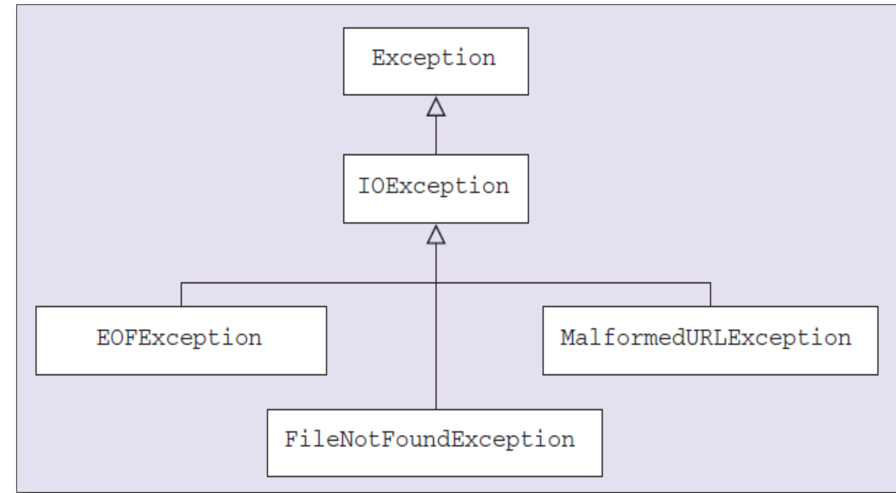


The figure shows a part of the exception class hierarchy

    - Errors that are described by the class Exception and its subclasses can be monitored and acted on. Some of these are defined as **checked exceptions**. Programmers **MUST include code** to declare or handle any checked exceptions that might occur. The Java compiler **will report an error** if this has not happened.
  - **(3) The `RunTimeException` class and its subclasses.**
    - These exceptions need not be caught or declared. They are called **unchecked exceptions.** Java contains facilities for catching the exceptions that can occur during the running of a program. are normally **due to programming errors** – these should be eradicated by proper design, good programming style and exhaustive testing.

3. Exceptions

- Examples of **checked exceptions:**
  - **`EOFException`** occurs when a program attempts to read past the end of a file.
  - **`FileNotFoundException`** can occur when a program attempts to open or write to an existing file, but the file is not found.
  - **`MalformedURLException`** indicates that an invalid form of URL (such as a website address) has occurred.



- Examples of **unchecked exceptions** (subclasses of **`RunTimeException`**):
  - **`ArrayIndexOutOfBoundsException`** occurs when a program tries to access an array with an index outside the valid limits.
  - **`ArithmeticException`** arises when an illegal arithmetic condition occurs, such as an attempt to divide an integer by zero.
  - **`NumberFormatException`** can be caused by an application expecting to read a number of some sort, when the data does not have the appropriate format.
  - **`NullPointerException`** happens when an application attempts to use null in a case where an object is required: for example, invoking a method on a null reference.

4. Declaring and handling exceptions

- For checked exception, you must do one of two things:

(1) **Catch the exception and deal with it within the method**, using a **try-catch** statement (**Handling exceptions**).

(2) **Declare in the method header** that an exception may be thrown

- in this case **you do not handle the exception within the method**, but simply **pass it on**;

4. Declaring and handling exceptions

(1) Handling exceptions

- To deal with an exception that has been thrown, we use a `try-catch` statement.
- Example: updating the previous code we wrote to read from a file

```
public static void main(String[] args) {
    try {
        File myFile = new File("C:/testing.txt");
        Scanner sc = new Scanner(myFile);
        String s = sc.nextLine();
        System.out.println(s);
        sc.close();
    } catch (FileNotFoundException e) {
        System.out.println(e);
    }
}
```

```
public static void main(String[] args) {
    try {
        File myFile = new File("C:/testing.txt");
        Scanner sc = new Scanner(myFile);
        String s = sc.nextLine();
        System.out.println(s);
        sc.close();
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

Notice the difference

it is best to be as specific as possible about the type of exception you expect to be generated. For example, it is usually a bad idea to do this in the above code.

- When this handler code is executed, the variable within the catch clause will contain a reference to the exception object that has been thrown.

4. Declaring and handling exceptions

(1) Handling exceptions (Cont'd)

- Example: If more than one type of exception is possible, we can add additional catch clauses, like this:

```
try
{
    if (checkFormat("File1.dat"))
    {
        // code to process file data
    }
}
catch (EOFException eofEx)
{
    // code to handle end of file exception
    // eofEx refers to an EOFException object
}
catch (MalformedURLException urlEx)
{
    // code to handle bad URL exception
    // urlEx refers to a MalformedURLException object
}
...
```

- The code in the try block is executed; if an exception is thrown the try block is exited and the appropriate exception handler code, if any, is executed. The exception handling code that is executed depends on which exception has been thrown.

4. Declaring and handling exceptions

(1) Handling exceptions (Cont'd)

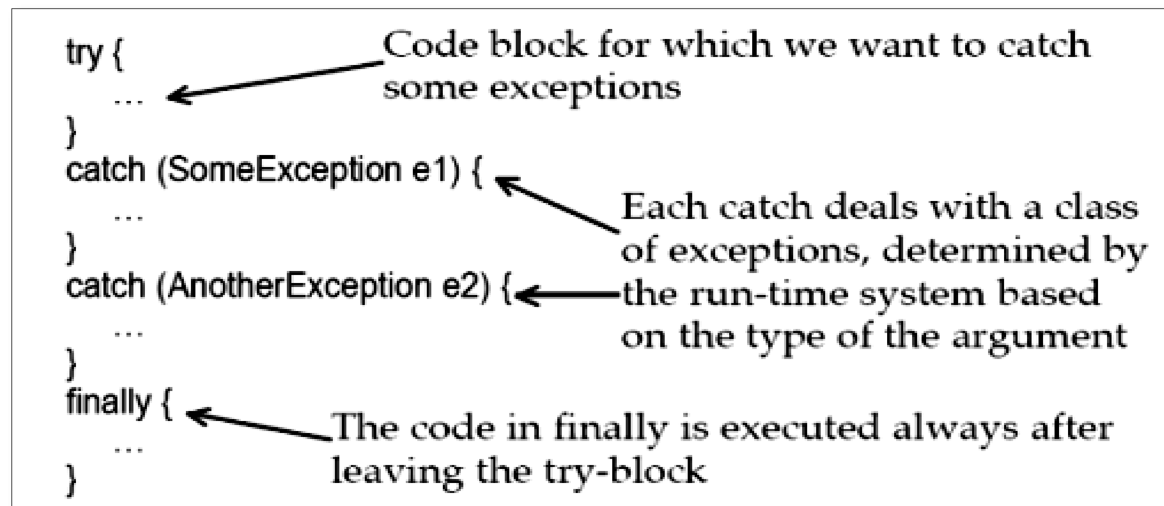
- It is possible and sometimes helpful to have more than one try statement within a method. This clearly defines the area of code where each exception is expected to arise:

```
try
{
    // code that may throw a FileNotFoundException
}
catch (FileNotFoundException ex)
{
    // code that handles a FileNotFoundException
}
//
// other code
//
try
{
    // code that may throw an EOFException
}
catch (EOFException ex)
{
    // code that handles an EOFException
}
```

4. Declaring and handling exceptions

(1) Handling exceptions (Cont'd) : the `finally` clause

- One of the things we often want to do after an exception has occurred is to 'clean up' by releasing any resources such as memory or files that a method has been using before the exception was thrown. This allows these resources to be used by other parts of the program or other programs.
- To deal with this, Java provides another facility in the `try` statement – the `finally` clause.
    - Code in the finally clause will be executed at the end of the try statement, whether or not execution was interrupted by an exception.



Source: http://developersbook.com/corejava/interview-questions/try-catch.png

4. Declaring and handling exceptions

(2) Declaring exceptions in the method header

- When declaring an exception in the method header, you use the keyword `throws` followed by the **type of exception** expected.

Examples:

Example1: If the method unexpectedly came to the end of the file when expecting to read more data, then the method would generate an exception of type EOFException.

```
public boolean checkFormat (String fileName)
    throws EOFException
{
    // code for method checkFormat
}
```

Example2: A method may be capable of generating more than one type of checked exception. In this case you need to list all the possible types

```
public boolean checkFormat (String fileName)
    throws EOFException, MalformedURLException
{
    // code for method checkFormat
}
```

Example3: If some or all of the possible exception types in a list are subclasses of a particular exception class, it may be possible to shorten the list.
•This works because `EOFException` and `MalformedURLException` are subclasses of `IOException`.

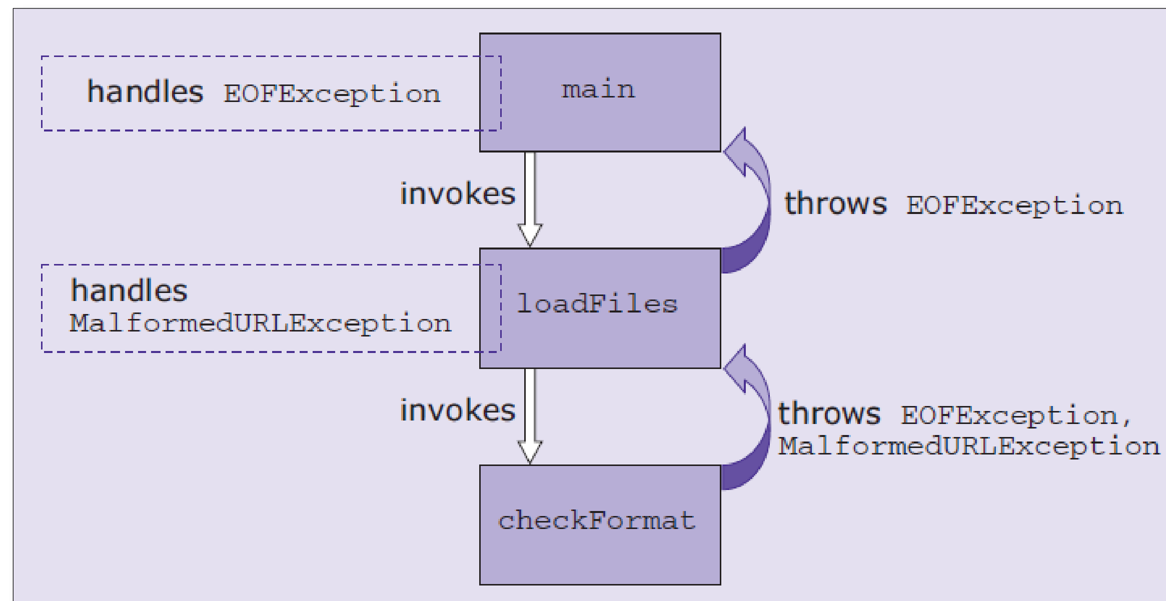
```
public boolean checkFormat (String fileName)
    throws IOException
{
    // code for method checkFormat
}
```

4. Declaring and handling exceptions

(2) Declaring exceptions in the method header (Cont'd)

What happens if the code throws an exception that is not handled in any try–catch statement within the method?

- Answer: then the exception will be passed to **the method that invoked this method**. Any exception not handled in a particular method will be **passed up a chain of invoking methods** to a higher level, until an appropriate exception handler is found. This is known as **propagating the exception**, as shown in figure.
- If **no handler exists** at any level for this exception, then the **program terminates** with an error message.



4. Declaring and handling exceptions

Creating exception classes

- An exception is just an object - in order to create one, create a new class that inherits from one of the **Exception** classes that are provided as part of the Java class library.
- Most exceptions have **two constructors**
  - a constructor with no arguments
  - and a one-argument constructor that allows for a more detailed error message.
- For example:

```
public class myException extends Exception {
    public myException(){
        super();
    }
    public myException(String s){
        super(s);
    }
}
```

5. Other error-handling techniques

Defensive programming

- There are techniques to deal with potential errors, which are based on anticipating the conditions under which errors will occur.
- For example,
  - to have a method return a value indicating whether the error condition was met when performing the code within the method.

IN the shown code, assume that we are writing a method called add, which adds an item to a queue of a fixed maximum size (e.g. array). Normally the add method would return no value and would just add the element to the end of the queue.

If the queue is already full, an error will occur. In order to cater for this error condition a Boolean return value could be used.

If the queue is full, then the method immediately returns the boolean value false and the new item is not added.

```
public boolean add (String newItem)
{
    if (isFull()) // private helper method isFull
    {
        return false; // queue full, item not added
    }
    else
    {
        // code to add item to end of queue
        // ...
        return true; // item added successfully
    }
}
```

5. Other error-handling techniques

Defensive programming compared to exception handling

- **Defensive programming**
  - It is appropriate when the potential error is **predictable** and **localized** .
    - e.g. , checking that a queue is not full before attempting to add a new element.
  - However, it has some **drawbacks.**
    - If there is a **lot of error checking**, this can obscure the main purpose of the method.
    - It is **ad-hoc** - different programmers may take different approaches to handling errors
    - It may not be possible for a method to return a value indicating that an error has occurred - for example, **if the method already returns a result**
- **Exception handling**
  - It is intended for conditions that are …
    - **Unpredictable:** caused by external events out of the control of the program, such as file errors
    - **Serious:** requiring the program to be terminated
    - **Widespread:** occurring at many places in the program, making it hard to check explicitly
  - Exception handling is typically much slower in execution than standard error-handling techniques - this is a reason to use it sparingly.

5. Other error-handling techniques

Example: dealing with a divide-by-zero problem (pseudo-code)

exception handling

```
try
{
   return (double) TotalRating / NumOfVotes;
}
catch (DivideByZeroException)
{
   return 0.0;
}
```

defensive programming

```
if (NumOfVotes == 0)
{
   return 0.0;
}
else
{
   return (double) TotalRating / NumOfVotes;
}
```