CS 244 Advanced Programming Applications

Dr Walid M. Aly

Exception & Java I/O

Lecture 5

Exceptions: Runtime Errors

```
1  class ExceptionDemo1
2  {
3  public static void main (String [] arg)
4  {
5  int x=9;
6  int y=x/0;
7  }
8  }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo1.main(ExceptionDemo1.java:6)
```

Information includes : class name-line number-exception class

```
1  class ExceptionDemo1
2  {
3  public static void main (String [] arg)
4  {
5  int []x= new int[3];
6  System.out.println(x[5]);
7  }}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at ExceptionDemo1.main(ExceptionDemo1.java:6)
```

```
1  class A{
2  void m(){}
3  }
4  class ExceptionDemo1{
5  public static void main (String [] arg)
6  {
7  m();
8  }
9  static void m()
10 {
11 A a1=null;
12 a1.m();
13 }}
```

```
Exception in thread "main" java.lang.NullPointerException
        at ExceptionDemo1.m(ExceptionDemo1.java:12)
        at ExceptionDemo1.main(ExceptionDemo1.java:7)
Press any key to continue . . .
```

The general form of an exception-handling block

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before
try block ends
}
```

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    }
  catch (ArithmeticException e) {
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

```
Division by zero.
After catch statement.
Press any key to continue . . .
```

- When an Exception is caught the program is transferred automatically to the appropriate catch block otherwise program stops
- The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.
- Finally is always executed .

Three Basic Rules for Exception Handling

1-Multiple catch Clauses

- After one catch statement executes, the others are bypassed.
- Exception subclasses must come before any of their superclasses, otherwise compile error

```
void m(int i){
int [] x={1,4,0,2};
try{
int d=4/x[i];}
catch(ArrayIndexOutOfBoundsException e)
{//}
catch(ArithmeticException e)
{//}
}
```

2-Exception Propagation

- If exception is not caught at its exact level , it propagates upwards towards the calling method
- Exception can be caught at a higher level

```
public static void main (String [] args){
ExcepDemo3 n=new    ExcepDemo3();
try{
n.m(3,0);}
catch(ArithmeticException  e){
//code to handle Exception}
}
}
void m(int x,int y){
int z=x/y;
}
```

java.lang

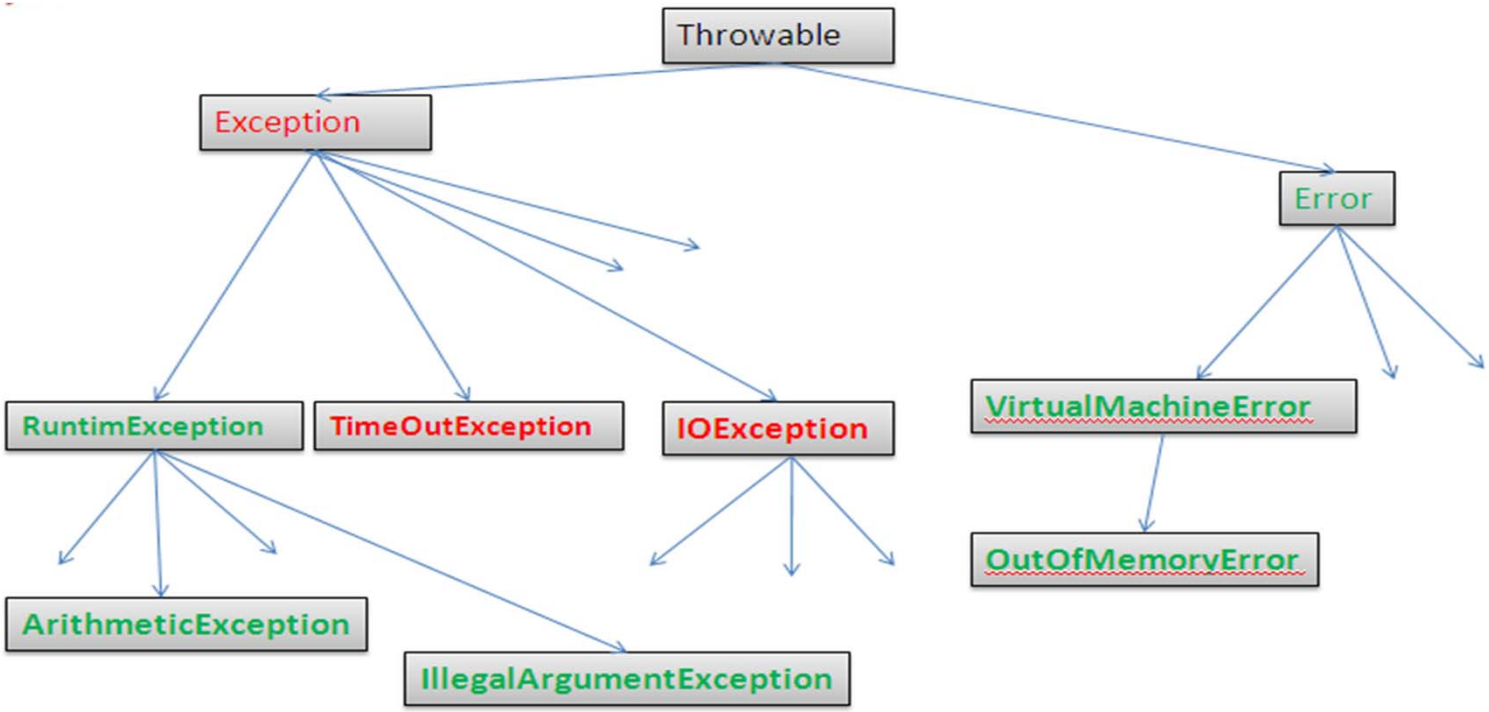
Class ArithmeticException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException
        └ java.lang.ArithmeticException

Polymorphism

3-You can Catch an exception using a reference of its super class

Exception Hierarchy



Exception Types

unchecked exception
could be unhandled by programmer

checked exception
should be handled by programmer

- Error and all of it subclasses are :unchecked Exceptions.
- RunTimeException with all its subclasses are unchecked exceptions
- Throwable , Exception and all its subclasses (except RunTimeException and its subclasses) are checked exception

Why Study Exception??

- Handling Unchecked Exceptions

  Although not mandatory , handling these exceptions will make your program reliable, robust and error prone.

- Handling checked Exceptions

  a lot of methods in the Java API throws checked Exceptions , especially

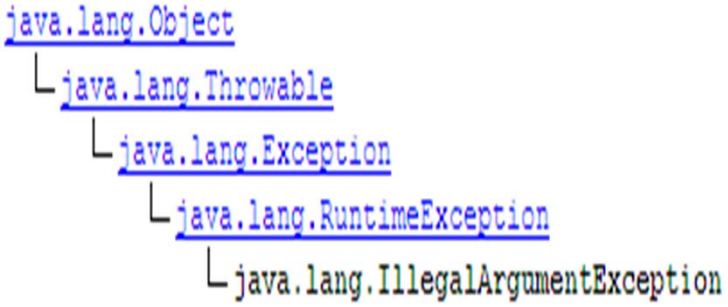
  – methods that reads from file.
  – Methods that connect your program to a remote P.C.
  – Methods that connect your program to a database (Connection and querying)

  You must handle these exceptions to use these methods

Using throws to declare the possibility of throwing exception

- A method declares the type of Exception that it might throws using the keyword throws
  Example : public void m() throws IllegalArgumentException

```
public void m() throws IllegalArgumentException
{
// a code that might throw IllegalArgumentException
}
```



- Method throwing checked exception is mandatory to declare it in method signature
- Method throwing unchecked exception is optional to declare it in method signature

Example : Handling The Exceptions from method Integer.parseInt

Class Integer

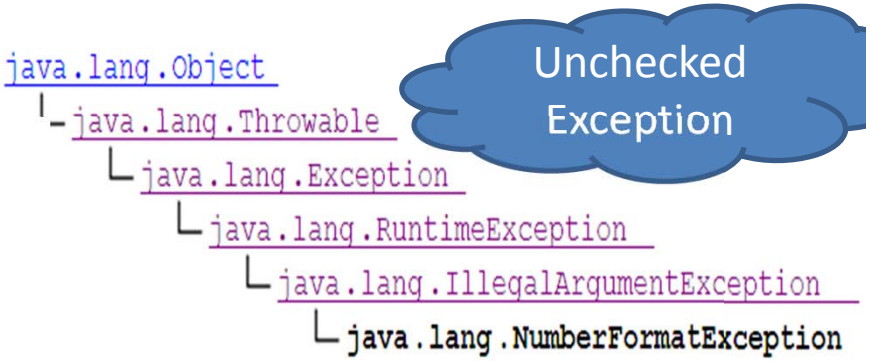
public static int **parseInt**(String s) throws NumberFormatException

Parses the string argument as a signed decimal integer .

……

Throws: NumberFormatException - if the string does not contain a parsable integer.

```
1   class ExcepDemo4{
2   public static void main (String [] args)
3   {
4   m("3");
5   m("a");
6   m("4");
7   }
8   public static void m(String number)
9   {
10  try
11  {
12  int i=Integer.parseInt(number);
13  System.out.println(i);
14  }
15  catch(NumberFormatException  e)
16  {
17  //code to handle Exception
18  }
19
20  }
```



Optional try-catch

```
3
4
Press any key to continue . . .
```

Creating Object of scanner to read data from a file

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

Constructs a new Scanner that produces values scanned from the specified file .

Parameters :

source - A file to be scanned

Throws :

FileNotFoundException - if source is not found

The calling method **should**

- surround the calling with try-catch
- or declare itself as throws,

Mandatory try-catch

java.io

Class FileNotFoundException

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

checked Exception

```
public static void m1 (){
File data=new File("Data.db");
try{
Scanner scan =new Scanner(data);
}
catch(FileNotFoundException  e)
{//code to handle Exception}
}
```

```
public static void m1 () throws FileNotFoundException
{
File data=new File("Data.db");
Scanner scan =new Scanner(data);
}
}
```

Throwing exceptions using throw

```
class Student
{
private int GPA;

public void setGPA(int GPA)
{
if(GPA>4)
throw new IllegalArgumentException();

this.GPA=GPA;
}
}
```

```
class Student
{
private int GPA;

public void setGPA(int GPA) throws Exception
{
if(GPA>4)
throw new Exception();

this.GPA=GPA;
}
}
```

- You can throw checked or unchecked exception using keyword throw
- Method that throws checked exceptions is **mandatory** to declare in its signature using keyword throws

Declaring your own exceptions

You can define new exceptions (checked or unchecked) by subclassing other Exceptions

```
class myException extends RuntimException
{

}
```

Unchecked Exception

```
import java.io.IOException
class myException extends I0Exception
{

}
```

checked Exception

Example

```
class GPAException extends RuntimeException
{

}
```

```
class Student{
private int GPA;
public void setGPA(int GPA)
{
if(GPA>4)
throw new GPAException();
this.GPA=GPA;
}
}
```

Using finally

- For each try-catch block there can be only one finally block.
- Case 1 :If an exception is thrown with a matching catch block then the **finally block is executed** after the catch block and program continues
- Case 2 : If an exception does not happen **the finally block will be executed** and program continues
- Case 3 : if an exception is thrown but not caught then **the finally block will be executed** after the try block , then program stops

```
try
{
//statment1
}
catch(exception e
{
//statment2
}
finally
{
//statment3
}
//statement 4
```

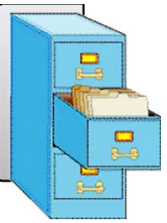
Case 1:statment1-statment 2-statment3-statment4

Case 2: statment1-statment3-statment4

Case 3:statment1-statment3

JAVA I/O Processing

Class File



- The `File` class in the `java.io` package represents files.
  - Create a `File` object to get information about a file on the disk. (Creating a `File` object doesn't create a new file on your disk.)

```
File f = new File("example.txt");
if (f.exists() && f.length() > 1000) {
    f.delete();
}
```

Package :java.io
File
Constructor
public **File**(String pathname)
Methods
public boolean **createNewFile**() throws IOException
public boolean **delete**()
public long **length**()
public boolean **exists**()

The File Class

- To operate on a file, we must first create a File object (from java.io).

```
File inFile = new File("sample.dat");
```

Opens the file sample.dat in the current directory.

```
File inFile = new File
        ("C:/SamplePrograms/test.dat");
```

Opens the file test.dat in the directory C:\SamplePrograms using the generic file separator / and providing the full pathname.

Some File Methods

```
if ( inFile.exists( ) ) {
```

To see if inFile is associated to a real file correctly.

```
if ( inFile.isFile() ) {
```

To see if inFile is associated to a file or not. If false, it is a directory.

```
File directory = new
        File("C:/JavaPrograms/Ch12");

String filename[] = directory.list();

for (int i = 0; i < filename.length; i++) {
    System.out.println(filename[i]);
}
```

List the name of all files in the directory C:\JavaProjects\Ch12

Some File Methods

```
if ( inFile.exists( ) ) {
```

To see if inFile is associated to a real file correctly.

```
if ( inFile.isFile() ) {
```

To see if inFile is associated to a file or not. If false, it is a directory.

```
File directory = new
        File("C:/JavaPrograms/Ch12");

String filename[] = directory.list();

for (int i = 0; i < filename.length; i++) {
    System.out.println(filename[i]);
}
```

List the name of all files in the directory C:\JavaProjects\Ch12

The **JFileChooser** Class

- A **javax.swing.JFileChooser** object allows the user to select a file.

```
JFileChooser chooser = new JFileChooser( );

chooser.showOpenDialog(null);
```

To start the listing from a specific directory:

```
JFileChooser chooser = new JFileChooser("C:/JavaPrograms/Ch12");

chooser.showOpenDialog(null);
```

Getting Info from JFileChooser

```
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
    System.out.println("Open is clicked");

} else { //== JFileChooser.CANCEL_OPTION
    System.out.println("Cancel is clicked");
}
```

```
File selectedFile  = chooser.getSelectedFile();
```

```
File currentDirectory = chooser.getCurrentDirectory();
```

Example

```
import java.io.*;
import javax.swing.*;

class FileChooserDemo
{
public static void main(String [] arg)
{
JFileChooser choose=new JFileChooser();
int status=choose.showOpenDialog(null);

if (status == JFileChooser.APPROVE_OPTION) {
            System.out.println("Open is clicked");
            File selectedFile  = choose.getSelectedFile();
            System.out.println("you choosed to open the file :"+selectedFile.getName());


} else { //== JFileChooser.CANCEL_OPTION
            System.out.println("Cancel is clicked");
}}}
```

Class Scanner

- To read a file, create a `File` object and pass it as a parameter when constructing a `Scanner`.

```
File f = new File("numbers.txt");
Scanner input = new Scanner(f);
```

java.lang.Exception
  └ java.io.IOException
    └ java.io.FileNotFoundException

Package :java.util
Scanner
Constructor
public **Scanner**(File source) throws FileNotFoundException
Methods
public String **next**() **Throws:**NoSuchElementException - if no more tokens are available
public double **nextDouble**()
public int **nextInt**()
public String **nextLine**()

Input tokens

- **token**: A unit of user input. Tokens are separated by whitespace (spaces, tabs, new lines).
- Example: If an input file contains the following:

```
23   3.14
  "John Smith"
```

  - The tokens in the input are the following, and can be interpreted as the given types:

Token	Type(s)
`23`	`int`, `double`, `String`
`3.14`	`double`, `String`
`John`	`String`
`Smith`	`String`

Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace.

Reading Data using Class Scanner

- A Scanner breaks its input into tokens using whitespace.(spaces, tabs, new lines).
- The resulting tokens may then be converted into values of different types using the various methods .

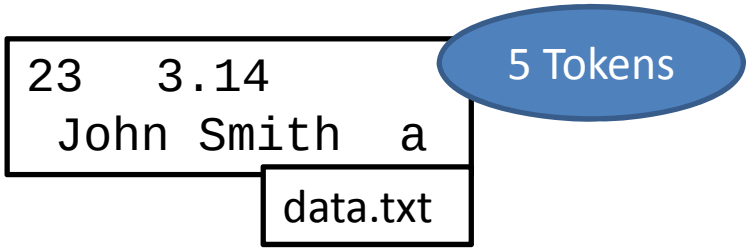
- Example: If an input file contains the following:

```
23    3.14    "John Smith“ a
```

  - The tokens in the input are the following, and can be interpreted as the given types:

Token	Type(s)
23	int, double, String
3.14	double, String
John	String
Smith	String
a	String

```
File file=new File(“data.txt”);
Scanner input=new Scanner(file);
int i=input. nextInt();
double d=input.nextDouble();
String s1=input.next();
String s2=input.next();
char x=(input.next()).charAt(0);
```



Example

Numbers.txt

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

Write a program that reads the first 5 values from this file and prints them along with their sum. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
Sum = 337.19999999999993
```

```
// Displays the first 5 numbers in the given file,
// and displays their sum at the end.
import java.io.*;   // for File, FileNotFoundException
import java.util.*;
public class Echo {
public static void main(String[] args)throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.txt"));
        double sum = 0.0;
        for (int i = 1; i <= 5; i++) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }}
```

Numbers.txt

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

public float **nextFloat**()

Scans the next token of the input as a float. This method will throw InputMismatchException if the next token cannot be translated into a valid float value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Float* regular expression defined above then the token is converted into a float value **Returns:**

the float scanned from the input

Throws:

InputMismatchException - if the next token does not match the *Float* regular expression, or is out of range

NoSuchElementException - if input is exhausted

IllegalStateException - if this scanner is closed

Testing before reading

- The preceding program is impractical because it only processes exactly 5 values from the input file.
  - A better program would read the entire file, regardless of how many values it contains.

- Reminder: The `Scanner` has useful methods for testing to see what the next input token will be:

Method Name	Description
`hasNext()`	whether any more tokens remain
`hasNextDouble()`	whether the next token can be interpreted as type `double`
`hasNextInt()`	whether the next token can be interpreted as type `int`
`hasNextLine()`	whether any more lines remain

Example

- Rewrite the previous program so that it reads the entire file. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
number = 4.7
number = -15.4
number = 2.8
Sum = 329.29999999999995
```

Numbers.txt

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

Example

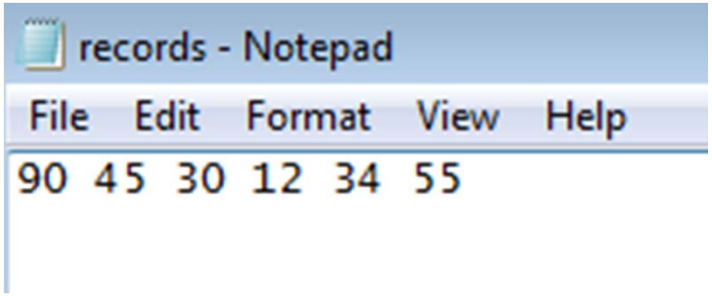
```
// Displays each number in the given file,
// and displays their sum at the end.
import java.io.*;
import java.util.*;

public class Echo2 {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }}
```

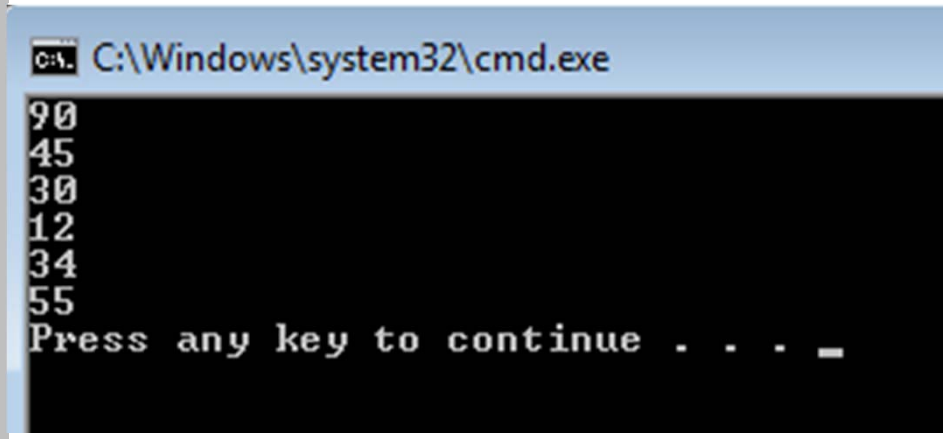
Numbers.txt

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

Example: Reading data from file

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;
class ReaderDemo{
public static void main(String [] arg){
File dataFile=null;
Scanner fileReader=null;
try{
 dataFile=new File("records.txt");
 fileReader=new Scanner(dataFile);
}
catch(FileNotFoundException exc)
{
System.out.println("Error  can not find file");
return;}

while(fileReader.hasNextInt()){
int i=fileReader. nextInt();
System.out.println(i);
}}}
```





Line-by-line processing

- Scanners have a method `nextLine` that returns from the input cursor's position to the nearest `\n` character.
  - You can use `nextLine` to break up a file's contents by line and examine each line individually.

```
while (input.hasNextLine()) {
    String line = input.nextLine();
}
```

Writing to files using Class PrintWriter

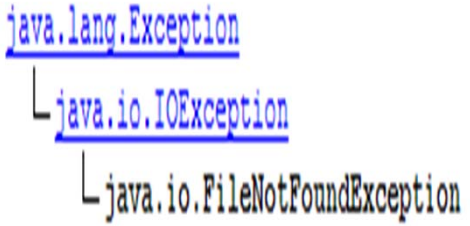
–If the given file does not exist, it is created.

–If the given file already exists, it will be truncated to zero size

```
PrintWriter output = new PrintWriter ("output.txt");
output.print("Hello");
Student ahmed=new Student(“Ahmed Aly”,32423);
output.print(ahmed);
```

N.B.: printing an object using System.out.println(object) or print(object) means automatically System.out.println(object.toString()) & print(object.toString())

Package :java.io
PrintWriter
Constructor
public **PrintWriter**(File file) throws FileNotFoundException
public **PrintWriter**(String fileName) throws FileNotFoundException
Methods
public void **print**(int i)
public void **print**(String s)
public void **print**(Object obj)



Printing to files, example

- Example:

```
PrintStream output = new PrintStream(new
  File("output.txt"));
output.println("Hello, file!");
output.println("This is a second line of
  output.");
```

  - You can use similar ideas about prompting for file names here.

- Do not open a file for reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - The result can be an empty file (size 0 bytes).
  - You could overwrite your input file by accident!

Web Resources

- Using Class RandomAcessFile
  - http://www.java2s.com/Tutorial/Java/0180__File/RandomAccessFileIntroduction.htm
- **Scanning text with java.util.Scanner**
  - http://www.java-tips.org/java-se-tips/java.util/scanning-text-with-java.util.scanner-3.html